Lehigh University
Lehigh Preserve

Theses and Dissertations

1985

Path planning and navigation for a mobile robot /

Sanjiv Singh
Lehigh University

Follow this and additional works at: https://preserve.lehigh.edu/etd

 Part of the Electrical and Computer Engineering Commons

Recommended Citation
Singh, Sanjiv, "Path planning and navigation for a mobile robot /" (1985). *Theses and Dissertations*. 4575.
https://preserve.lehigh.edu/etd/4575

This Thesis is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.

PATH PLANNING AND NAVIGATION FOR A MOBILE ROBOT

by

Sanjiv Singh

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Electrical Engineering

Lehigh University

1985

Certificate of Approval

This thesis is accepted and approved in partial fulfillment of the requirements for the degree Master of Science in Electrical Engineering.

July 31, 1985

(date)

Professor in Charge

Chairman of Department

Acknowledgments

Credit is due to Dr. Wagh who was my mentor. His helpfullness gave me the strength for repeated revisions. His insights have been the biggest external influence on this work. Dr. Eberhardt has been the source of the theory used to develop the Navigation algorithms as well of the scientific pragmatism without which this project would never have got as far as it has. I would also like to thank Dr. Nagel for his initial suggestions that resulted in the work on Path Planning and all the people who took time to review this thesis.

The author was supported by a research project funded by a Ben Franklin project and SI Handling.

Table of Contents

List of Figures

List of Tables

Abstract

Automated Guided Vehicles are expected to be used increasingly in the factory of the future. Two important areas of R&D lie in Path Planning and Navigation for these vehicles. This thesis presents methods to automatically plan and navigate paths for a prototype mobile factory robot which has been developed at Lehigh.

The navigation strategy presented describes a control structure that can locate a mobile robot in cartesian coordinates and guide it along the specified path. The path planning strategy identifies *all* rectangular "convex" areas (those areas inside which travel is possible in a straight line) in a given environment. A graph is created where the convex shapes isolated are represented as nodes and their intersections as arcs. Since fixed costs cannot be associated with arcs of this graph, a new backtracking graph traversal technique is developed that dynamically allocates costs to a developing path. The paths thus obtained are optimal is most cases, requiring only minimal refitting. Further, if the obstacles are aligned (as is likely in an industrial situation), the computational complexity of the algorithm decreases considerably.

Chapter 1
Introduction

The technology to automatically make a vehicle (such as a mobile factory robot) traverse a specified path has existed for quite sometime. However, such technology has not been implemented in industry because navigation systems required to maintain the repeatibility requirements have been cost prohibitive. Thus we have seen automatic devices custom built for specific purposes. Wire guided vehicles are a prime example. These machines. although functional, are severely limited in terms of flexibility. This thesis suggests a structure and method of operation for a vehicle that would meet the need of economic feasibility, flexibility, and accuracy required in industry.

Essentially, the work done for this thesis was focused on two areas:

- The development of a system responsible for guiding a mobile factory robot along a specified path.
- The development of a strategy to automatically plan paths for mobile robots in relatively structured and non-volatile environments.

The first of these areas, described in Chapter 2, can be conceptually divided into considerations of two concurrent tasks:

Self Location- responsible for locating the robot in cartesian coordinates and bearing at any given moment. This is achieved by two inter-dependent navigational sub systems. A gyroscope and a wheel encoder constitute the primary navigational unit and provide input to instantly locate the vehicle as it moves. A laser scanner is used as a secondary navigational unit to obtain a "fix", an accurate measure of vehicle orientation and position and to correct the error built up in the primary system.

Navigation- responsible for directing the movement of the robot by controlling the steering angle and speed necessary to track the path specified by a list of points and starting and ending orientations of the vehicle. This is achieved by first identifying vectors that represent vehicle position and orientation at points spread out regularly along the specified path. Initially the navigational parameters are computed assuming the vehicle will travel on a smooth path between specified points. If the vehicle strays from the ideal path. its steering angle and speed are recomputed to bring it back on track.

Fig. 1-1 shows a simplified block diagram of the vehicle "Cyclopian" incorporating this dual navigation scheme.

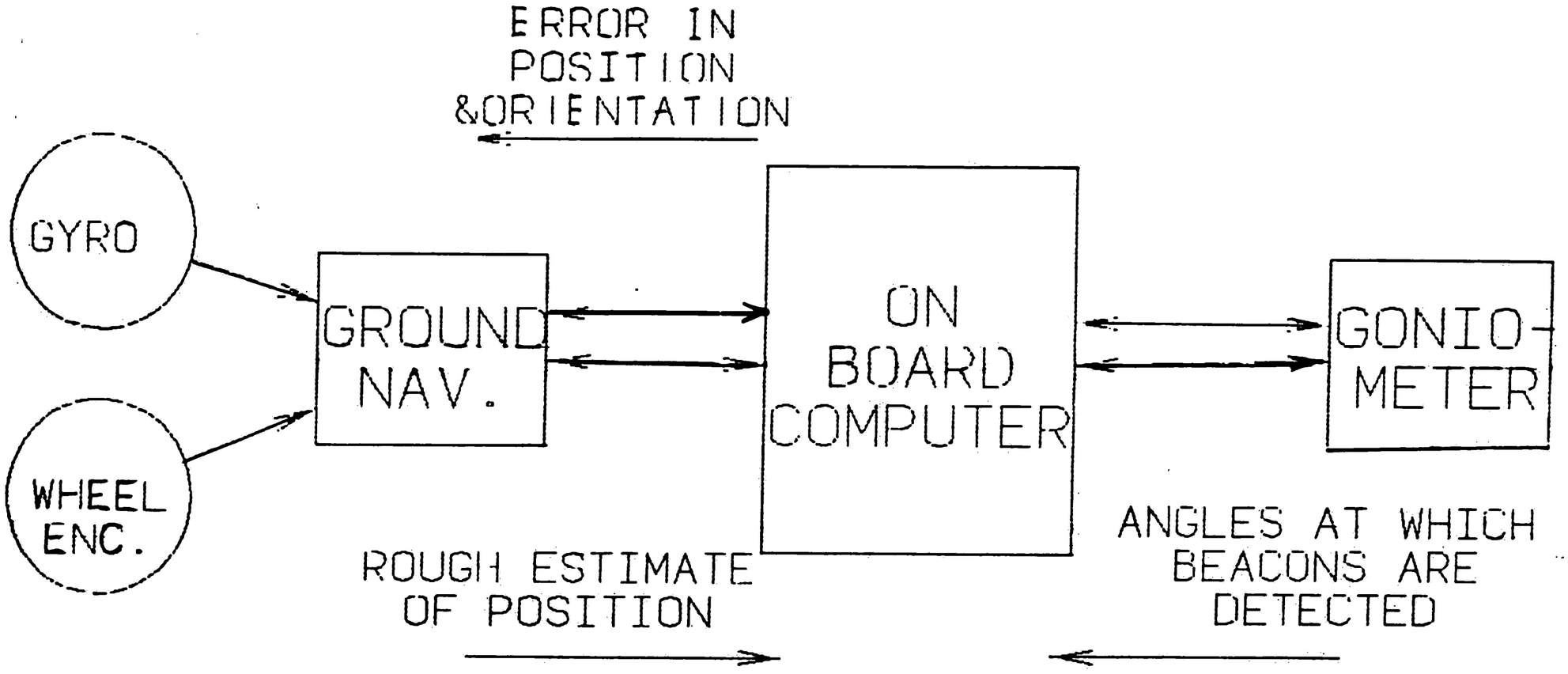


Figure 1-1: The Navigation System of "Cyclopian"

"Cyclopian" was built as a prototype of an industrial mobile robot to work out some out the primary problems involved in this project. "Cyclopian" is a three wheeled vehicle where both the driving and steering are done by the front wheel. At present it runs on three car batteries. The hardware onboard the vehicle includes a single board computer (Intel 86/05), a gyroscope. an infra red

laser scanner, and coding disks on the front wheel.

The second area of concentration described in Chapter 3, considers the automatic path planning for a mobile robot [1]. Currently, there are two philosophies regarding path planning for robots. The first is the academic, more general approach, the proponents of which are interested in suggesting schemes that are as flexible and generally applicable as possible. Their approach assumes arbitrary shape, size and orientation of the robots and obstacles [2, 3]. Consequently the solutions obtained have a complexity exceeding that required of a system to function in structured non-volatile industrial environments. The second philosophy, espoused especially by the industry, is to construct a workable system albeit with severe limitations. For example a robot might only have knowledge of how to get back and forth between work stations; if it were required to travel to a new point, a new path would have to be perhaps hand computed or picked by inspection [4]. This approach has worked in industrial situations where the environment and tasks allocated to robots are not likely to change much at all.

In a sense, this thesis reflects a compromise between the two approaches described above. The navigation system meets the need of being able to navigate a path autonomously without the necessity of external aids such as a wire buried in the ground. The path planning strategy automatically provides optimal paths given only arbitrary starting and destination points and a map of the obstacles. These techniques make assumptions about the environment and expected performance of the vehicle. While these assumptions prohibit the techniques from being used for a general purpose mobile robot, they are within the bounds of acceptable parameters in industrial situations.

The algorithms of Chap. 2 and 3 are implemented in Pascal on a DEC 2065 computer. Implementation details can be found in [5].

Chap. 4 presents some of the simulation results pertaining to navigation and path planning algorithms. Finally, Chap. 5 presents conclusions and makes suggestions for possible extensions to this work.

Chapter 2
Robot Self Location and Navigation

This chapter considers the negotiation of a specified path by a mobile robot. The principle used to locate the robot in a cartesian plane is similar to that used by bats to navigate. By sending out signals (in our case optical signals rather than the high frequency sound signals used by bats) and then sensing their reflections from the surroundings, the robot vehicle is able locate its position relative to obstacles. Since the scope of this system is limited to industrial settings, it is feasible to lay a fixed coordinate system on the factory floor. Given that the location of the obstacles is known accurately, the position of the robot can also be pinpointed.

Once a reasonable estimate of the vehicle position is guaranteed, the next problem to be tackled is navigation/driving the vehicle along a smooth path specified by discrete points. The robot is instructed to follow arcs between such points. At the end of each arc, based upon the actual location of the vehicle, a new path is computed to the next point. This compensates for the deviation from the ideal path caused by sluggishness of the steering angle change and wheel slippage.

2.1 Self Location using the Goniometer and Ground Navigator

A fundamental problem is to accurately locate the vehicle at any given point. This is accomplished by two interdependent navigational systems: A Ground Navigator to keep a rough estimate of the position and a Laser Scanner or **Goniometer**) to periodically refine that estimate. Each system is vital to robot self location. Whereas the accuracy is provided by the goniometer, the

neighborhood data is provided by the ground navigator.

2.1.1 Obtaining a "fix" from the Goniometer

The Goniometer is a device that emits a circularly sweeping infrared laser beam in a plane parallel to the ground. It is mounted on the vehicle at the mid point of the rear axle[1]. It is able to detect the angles at which "beacons" that are posted around the room, are spotted. "Beacons" are retro-reflectors which are fabricated in such a way that they reflect light at the incident angle (the light beam is reflected back on the same path on which it arrived). Every time a reflection is detected, a number corresponding to the angle it was detected at, is stored in a buffer. This buffer is initialized every time the beam has swept a complete circle. Since the beam sweeps a circle every half a second, it is possible to poll the buffer twice a second for the angles at which the beacons were noticed.

The computer has, at any given time, access to at least a rough estimate of vehicle position by polling the ground navigator. After a "fix" has been given by the goniometer, the ground navigator is recaliberated, but errors progressively build up with time. Using a map of beacon locations, and the angles from the buffer described above, the LOCATE algorithm (described below) matches the angles at which beacons were seen to angles at which beacons are expected. Through geometrical calculations the position of the vehicle can be determined.

---

[1]This point has been chosen because it results in the simplest mathematics. In principle the Goniometer can be located anywhere on the vehicle.

2.1.2 Efficiently matching the beacons

Given that the vehicle is at rest and that angles between beacons are spread out when a goniometer reading is taken, locating the vehicle is straight forward. Assuming that the angles at which beacons were spotted are accurate, the problem in essence is to match the beacon sightings with a map of the beacons. In a realistic situation it is necessary to consider the fact that there may be many beacons to match as well as the fact that positional errors obtained by using readings of beacons far away are likely to be greater. The need to "efficiently" locate the vehicle is addressed by intelligently picking good beacon sightings. This problem is tackled by also storing a reading that corresponds to the apparent angular width of the beacon spotted. Notice that this is an approximate measure of distance between the vehicle and beacon.

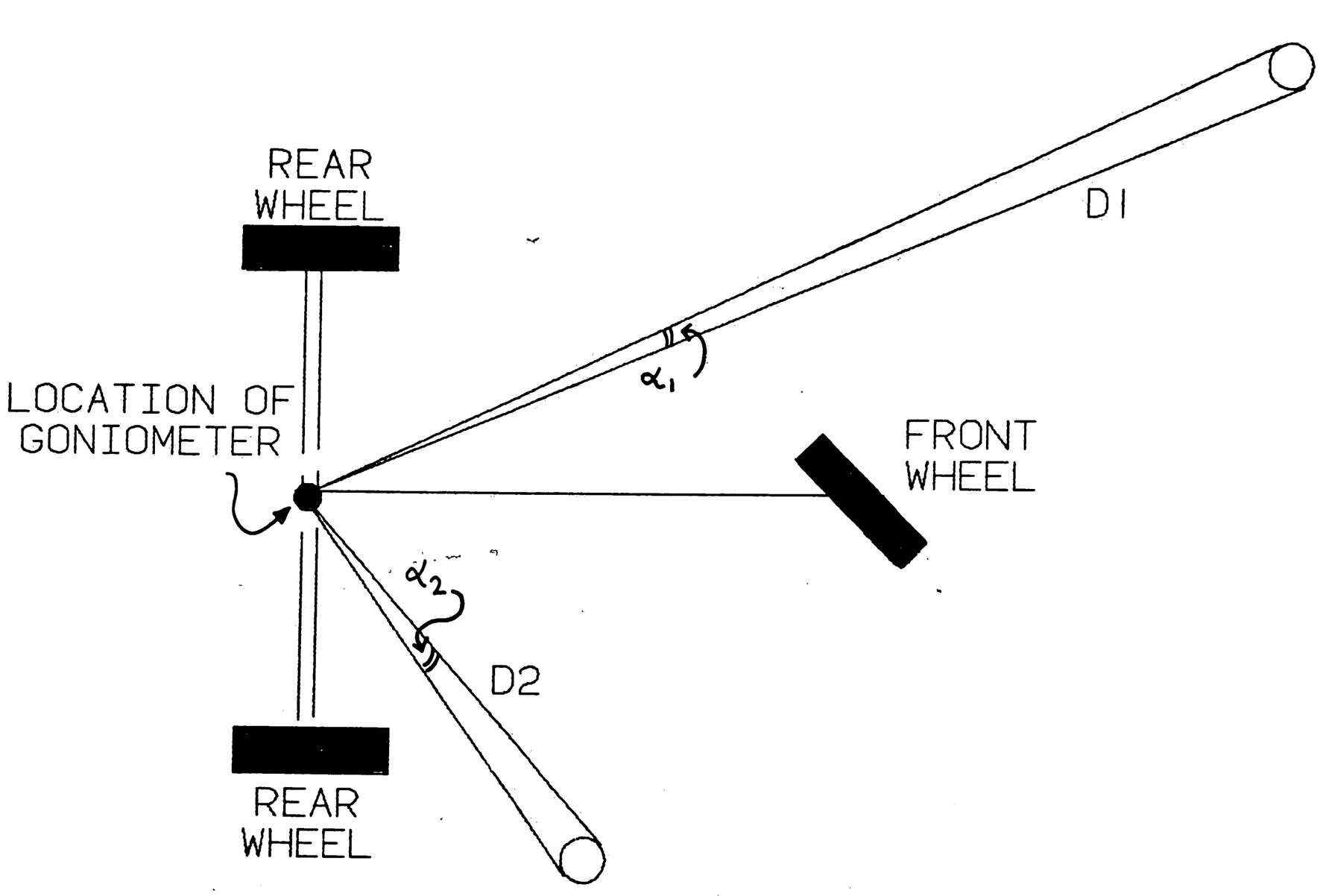


Figure 2-1: Measuring Distance to Beacons

Fig. 2-1 shows how the distance d is inversely proportional to the angle a. Using this approach also necessitates that beacons have the same apparent width from all directions. This need is met by making the beacons into cylindrical posts. Now, when beacon angles are detected, the ones that have angular widths smaller than a chosen threshold, are discarded. Notice that spurious reflections (assuming that they have a small angular width) will be discarded immediately since they will appear very far away.

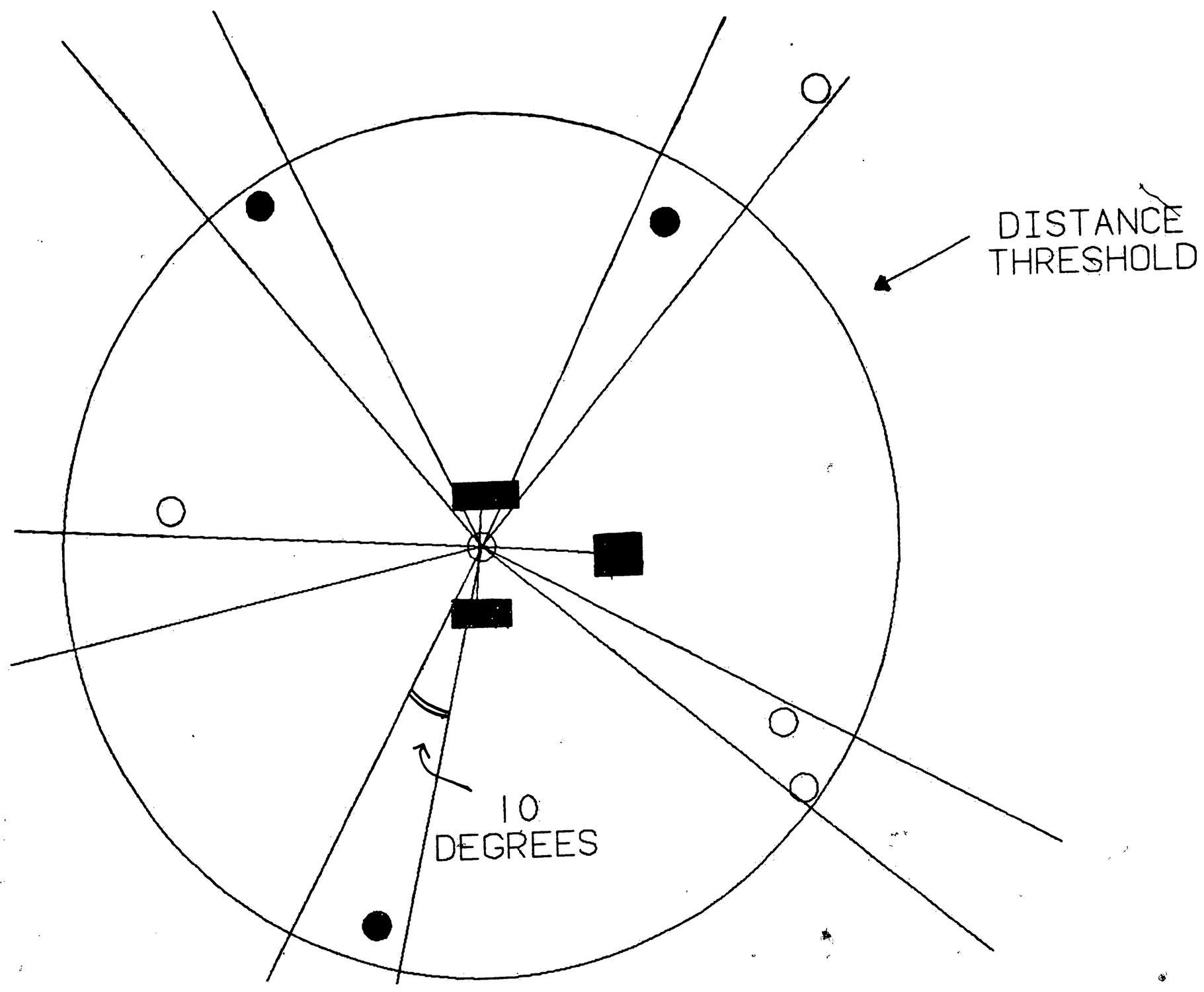


Figure 2-2: "Efficiently" matching sighted beacons

Because of the uncertainty of vehicle position and orientation, beacons have to be looked for in a cone emanating from an angle equal to the largest

error expected. We use a cone of 10° corresponding to a ± 5° error. There may be more than one beacon within the angular range but most times this will not cause a problem because the distance criterion will discard those beacons that are far. However, if there is still more than one beacon with in a cone, both of them are discarded because their is no way of telling which of those corresponds to the beacon spotted. If only one beacon is found in the cone, then the x,y position of that beacon (from the map) is added to a list of matched beacons. Fig. 2-2 shows the beacons that are picked (shaded dark) and those that are discarded (unfilled). If at least 4 beacons cannot be matched then the distance threshold is extended. From every triplet of beacons that can be made from this list, one computation of vehicle position can be made using the LOCATE procedure as described below:

Let the cartesian coordinates of the three beacons be denoted by (X_1,Y_1), (X_2,Y_2), (X_3,Y_3). Let the unknown point (the point to be determined) be (X_0,Y_0). Let the angle between beacons 1 and 2 be by α_1, and the angle between beacons 2 and 3 be α_2.

1. Draw two circles A & B such that (X_0,Y_0), (X_1,Y_1), and (X_2,Y_2) lie on circle A and (X_0,Y_0), (X_2,Y_2), and (X_3,Y_3) lie on circle B (Fig. 2-3).

2. Find C_A, C_B the centers of circles A and B (Fig. 2-4). The center of each circle can be found in the following manner:

$$m_1 = \frac{Y_2 - Y_1}{X_2 - X_1} \qquad m_2 = \frac{X_2 - X_1}{Y_1 - Y_2} \tag{2.1}$$

$$C_x = x' + \frac{l \cdot cot(\alpha)}{\sqrt{1+m_2^2}} \qquad C_y = y' + \frac{l \cdot cot(\alpha) \cdot m_2}{\sqrt{1+m_2^2}}$$

Since the above formulas can result in 2 different solutions (the other is the mirror image of the center) for $C_{x,y}$, the combination that is equidistant

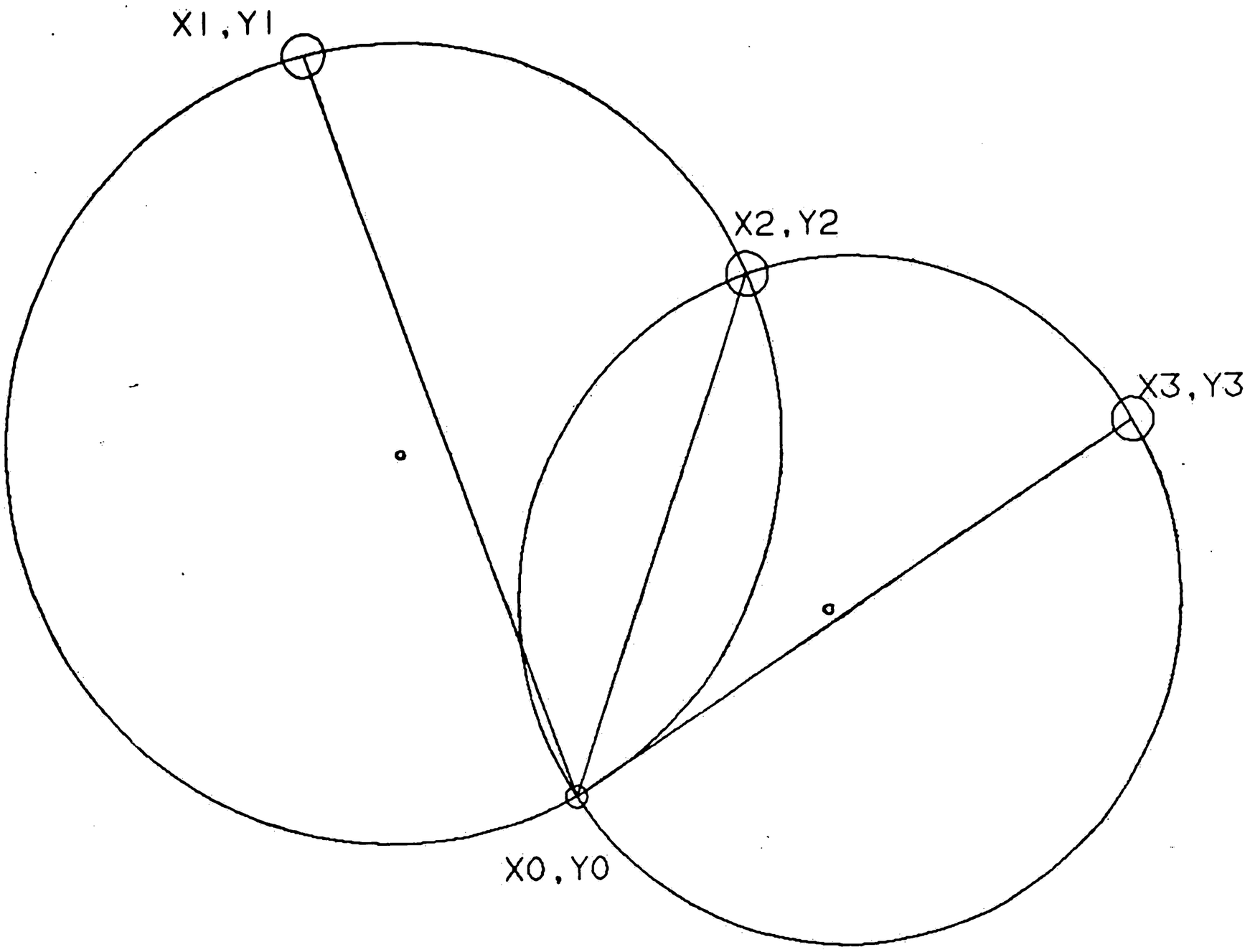


Figure 2-3: Placing a beacon triplet on two circles

from the three points on the circle, is selected as the center point.

3. Locate the point (X_m, Y_m) in the following manner: m, m' are defined as in Fig. 2-5.

$$m = \frac{C_{y_B} - C_{y_A}}{C_{x_B} - C_{x_A}} \qquad m' = \frac{C_{x_B} - C_{x_A}}{C_{y_A} - C_{y_B}} \tag{2.2}$$

$$X_m = \frac{X_2 \cdot m' - C_{x_A} \cdot m + C_{y_A} - Y_2}{m' - m} \qquad Y_m = (X_m - X_2) \cdot m' + Y_2$$

4. Find (X_0, Y_0):

$$X_0 = 2 \cdot X_m - X_2 \qquad Y_0 = 2 \cdot Y_m - Y_2 \tag{2.3}$$

Once (X_0, Y_0) has been pinpointed, the bearing ϕ can be found using any one

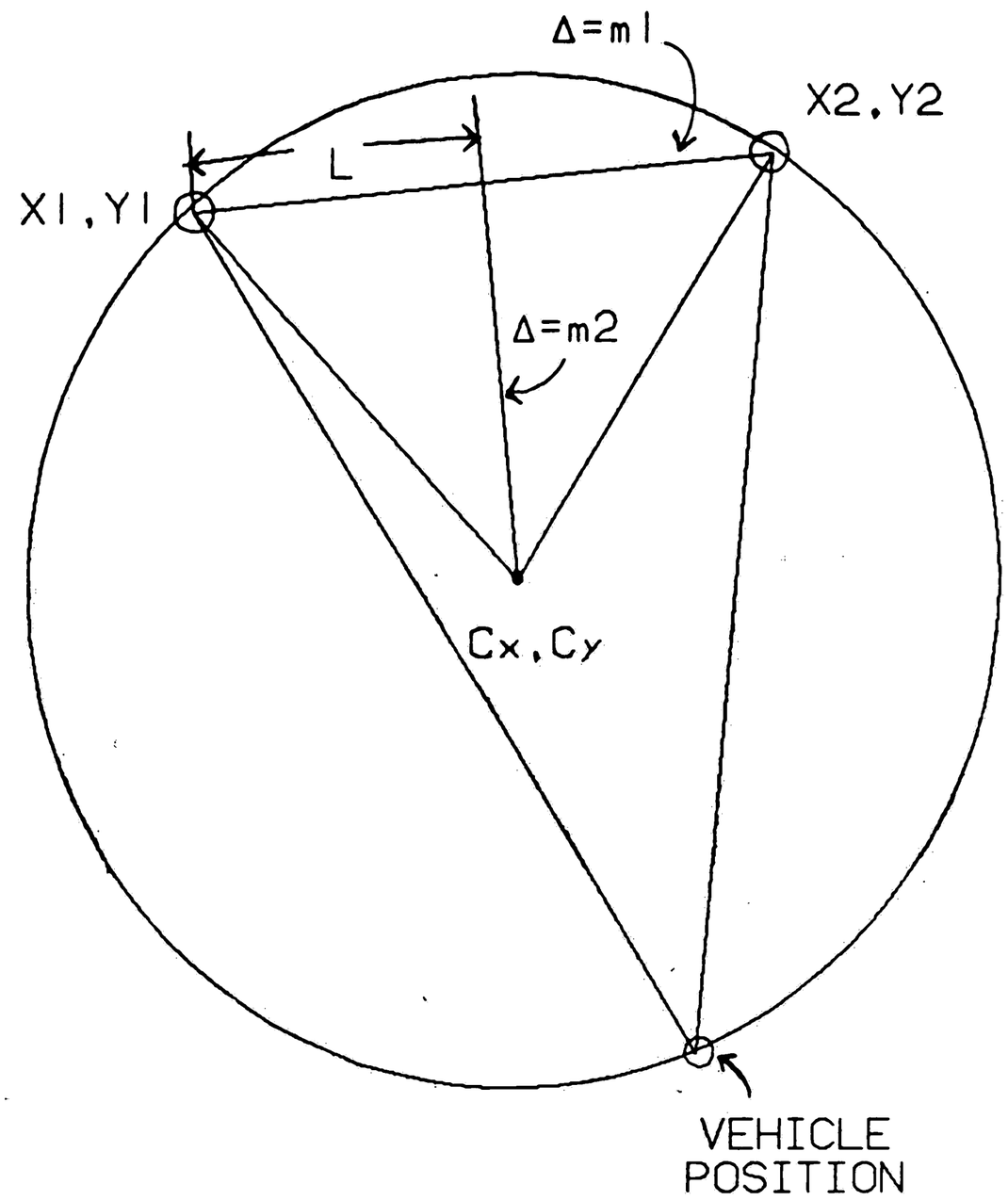


Figure 2-4: Finding the center of each circle

of the beacons. It should be noted that since calculation of the vehicle position depends on drawing two circles such that the first beacon, second beacon and the point to be located (X_0,Y_0) lie on one circle, and, the second beacon, third beacon and (X_0,Y_0) lie on a second circle, this method will fail if all three beacons lie on the same circle. However, since a large number of triplets can be made from a relatively small number of beacon sightings, this case, should it ever arise, is simply discarded.

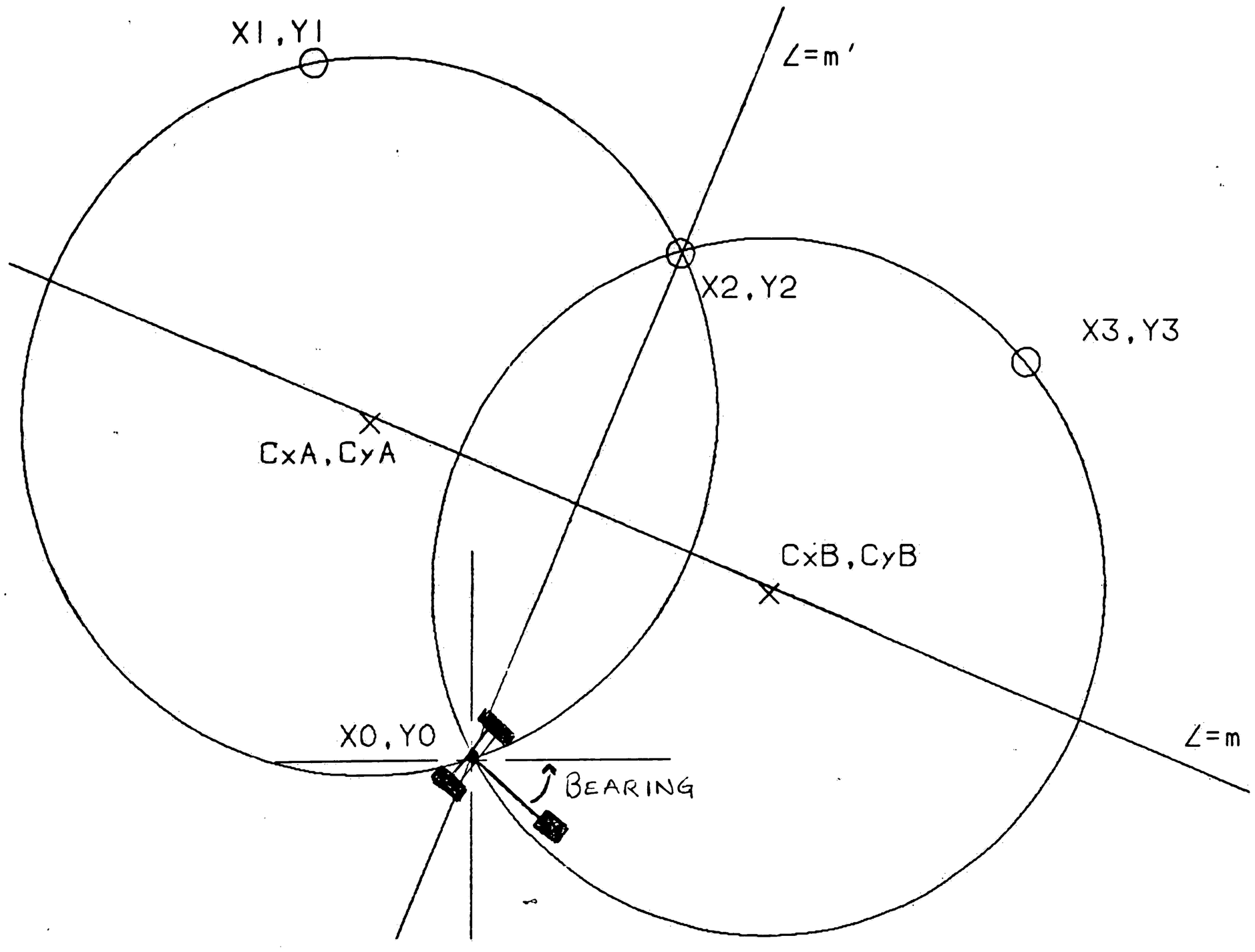


Figure 2-5: Locating the Vehicle Position and Orientation

2.1.3 Compensating for vehicle movement

If the vehicle is moving at all while the goniometer reading is taken, we have the problem that the $i+1$-th beacon is spotted from a different ground location than the i-th beacon (Fig. 2-6). Notice that this problem can be compensated for by slowing down to rest every time a goniometer reading is to be taken. The other alternative is to store data from the ground navigator into a buffer to correspond with beacon sightings, effectively recording the position from where each beacon was spotted. Hence, every time a beacon is sighted, the following is stored:

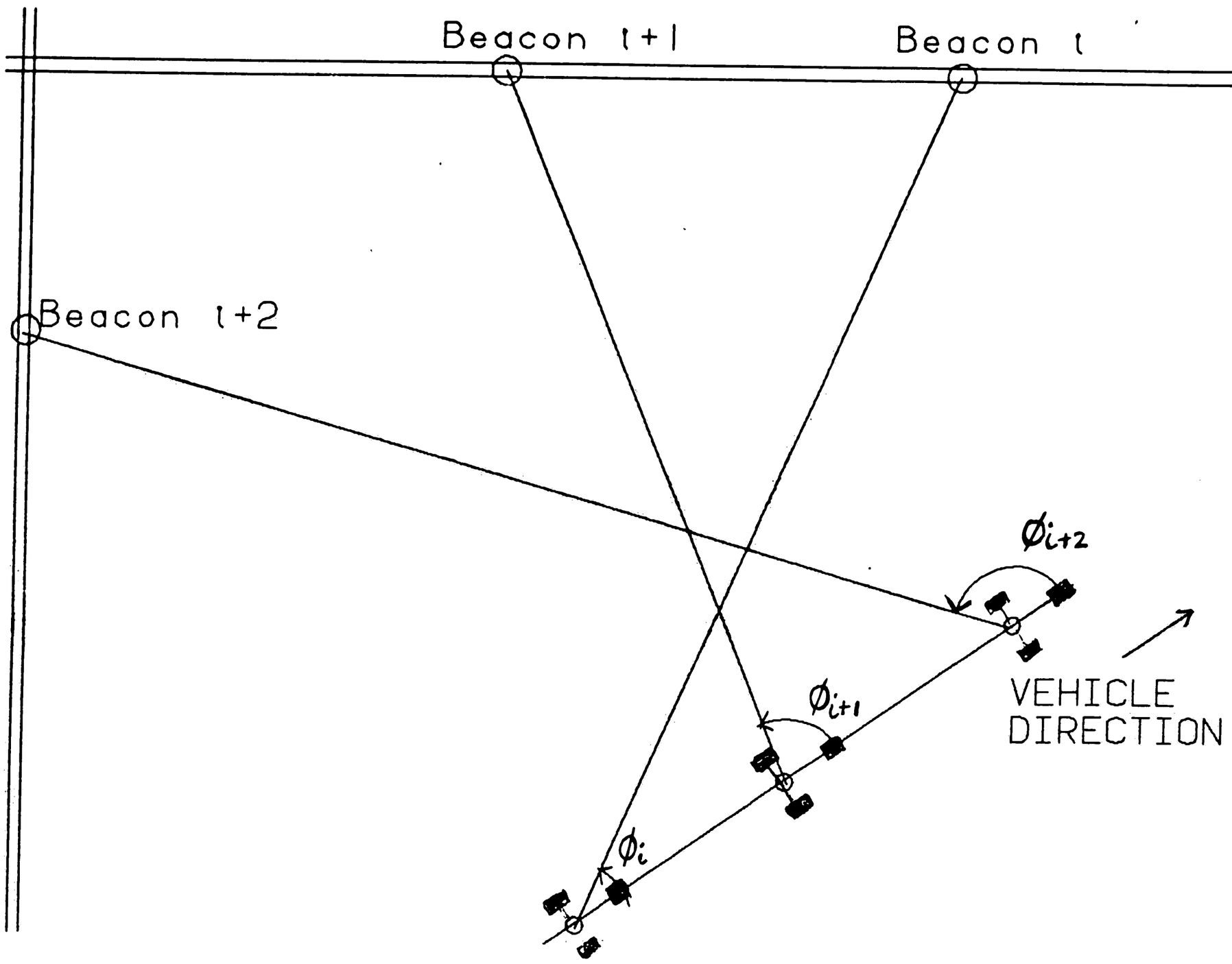


Figure 2-6: Beacon Sightings from a Moving Vehicle

1. Angle at which the reflection is detected.

2. A measure of distance between the vehicle and the beacon

3. Ground Navigator data corresponding to the floor position at which the beacon is sighted.

Every time a "fix" is obtained, we would like to actually determine exactly how far off the ground navigator is from the true orientation and position, as opposed to absolute location. We assume that the difference between the true location and the location as computed by the ground navigator (error built up in the ground navigator) does not change significantly in the time required for the computation. This error term can be obtained by comparing the ground navigator location to the location calculated by the goniometer at the *same*

instant. It is convenient to do this at the exact moment when an INDEX PULSE is issued. The INDEX PULSE is a signal sent by the Goniometer every time it completes one rotation. Using the INDEX PULSE, the ground navigator record the floor position that corresponds to the location of the vehicle at that instant. When a "fix" is obtained, the position and the orientation of the vehicle computed is compared with the values stored at the instant of the INDEX PULSE to obtain the error in the ground navigator.

The problem is thus reduced to modifying the beacon sightings so that they are in reference to the robot position at the arrival of the INDEX PULSE (Fig. 2-7).

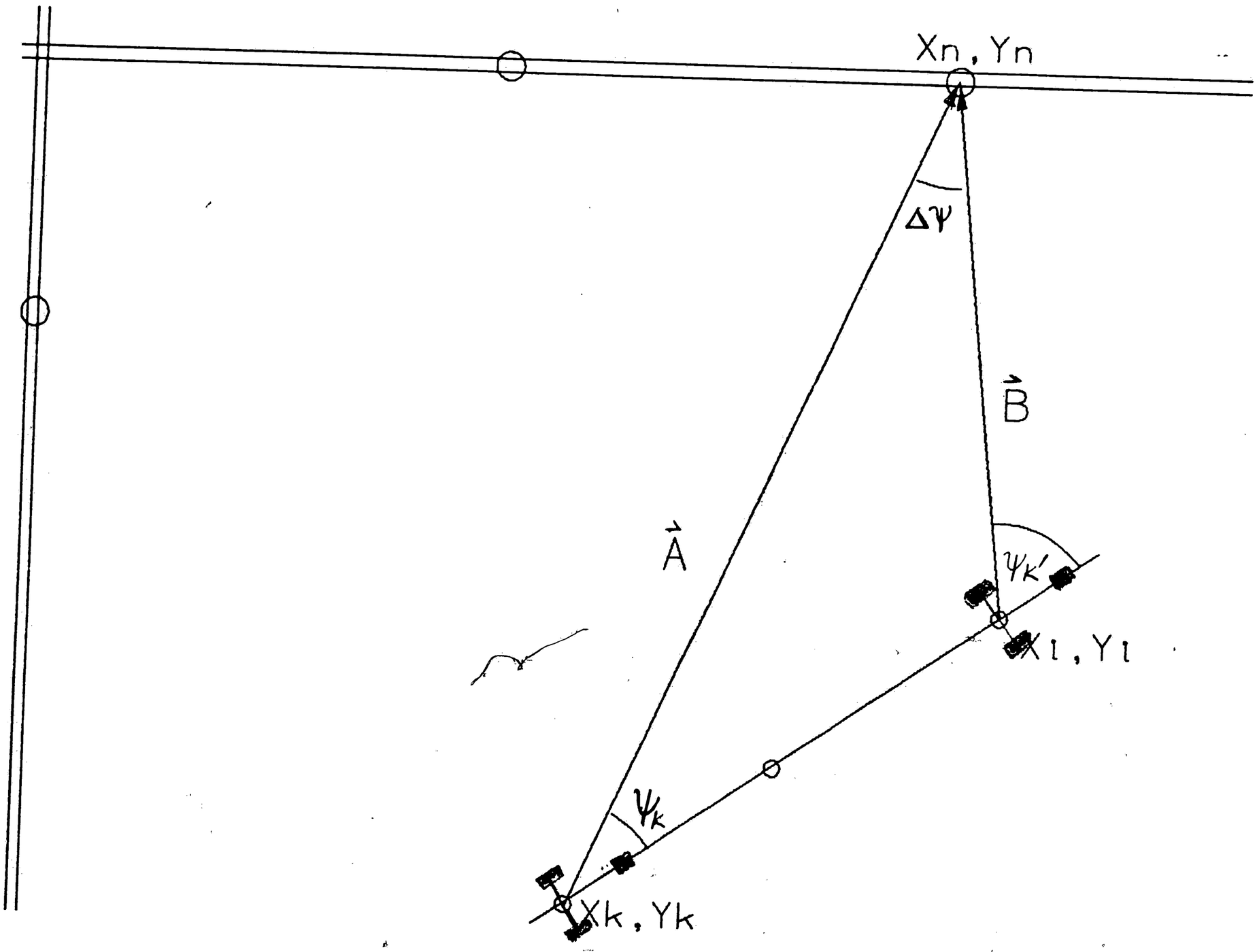


Figure 2-7: Making compensations for vehicle movement

This modification may be described as follows: For each beacon sighting the following adjustment is made: Let a beacon at (X_n,Y_n) be sighted at angle ψ_k when the vehicle is at (X_k,Y_k). Further, let the vehicle position when the)$_k$. Let the position at which the INDEX PULSE is issued be (X_i,Y_i). Since we would like all beacon angles to be relative to the vehicle position at this point, we need to find ψ_k' by determining $\delta\psi$:

$$\delta\psi = \sin^{-1}\left[\frac{(X_n - X_k)\cdot(Y_n - Y_i) - (X_n - X_i)\cdot(Y_n - Y_k)}{\sqrt{|(X_n - X_k)^2 + (Y_n - Y_k)^2|}\cdot|(X_n - X_i)^2 + (Y_n - Y_i)^2|}\right] \tag{2.4}$$

$$\psi_k' = \psi_k + \delta\psi$$

Notice that the correction of angle sightings is dependent on the length of the vectors $\overline{A}$ and $\overline{B}$ which we can only know to an approximation. In the ideal case, a "fix" is obtained when the vehicle is at rest in which case the error due to approximating the two vectors goes to zero. This method of compensating for the vehicle is used for the lack of a better method at present.

2.1.4 Keeping a rough estimate through the Ground Navigator

In theory it is possible to keep track of the vehicle position by means of only a gyroscope and wheel encoders. As a matter of fact, if the Goniometer is not able to match enough beacons, the vehicle has the capability of traveling guided solely by the Ground Navigator. The gyroscope can indicate the differential in angular positioning of the vehicle while the wheel encoders can provide information of how far the wheel has moved. However, there is slippage in the wheels. and unless an extremely sophisticated gyroscope is used, significant angular errors associated with the earth's rotation build up resulting in a progressively degraded estimate of vehicle position.

The goniometer as described above is used to periodically correct (calibrate) the gyroscope by giving an accurate "fix" of vehicle position and orientation. This is possible because over small time intervals the ground navigator provides reasonable estimates in differentials of x-y movement. It should be noted that the ground navigator is essential; the goniometer could not work without it. Since the scheme used cannot uniquely identify the beacons merely from the reflection data (all the beacons are identical), it must have at least a rough idea of where the vehicle is located to be able to match the beacon sightings with the beacon map.

The Gyroscope

The gyroscope used has an inherent deviation of 4 degrees per hour. It provides a differential output corresponding to the angular position of the vehicle. Since it doesn't provide an absolute angle position, every time a goniometer reading is taken, the zero degree position has to be recaliberated. This can be done by keeping an offset value in the program which corresponds

to the difference in the output of the gyroscope and the true angle. Once again, if we assume that the gyroscope provides reliable data over small time intervals, then it is possible to obtain a good estimate of angular position, providing the offset is recaliberated at reasonable intervals.

The Wheel Encoders

The wheel encoders are set up so as to provide a pulse every time the front wheel travels through a distance of 1/3 inch. Given a top speed of 4 ft/sec. this corresponds to a maximum of 50 pulses per second.

2.1.5 Computing Ground Position Using the Ground Navigator

The ground position could be computed from the differential gyroscope and wheel encoder thru a simple integration (Fig. 2-8):

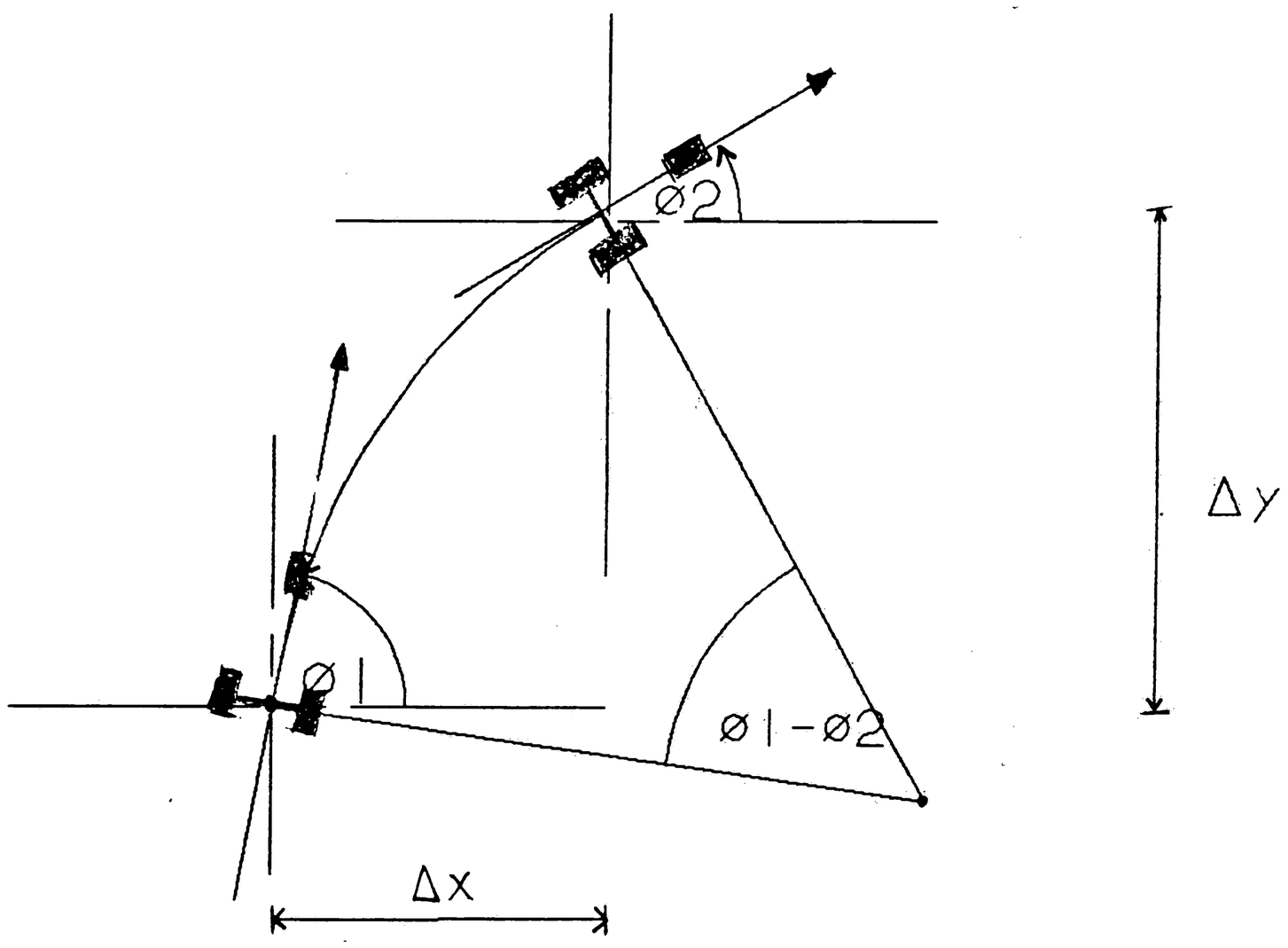


Figure 2-8: Ground Navigator Calculation of Differential Vehicle Movement

If the length of a small arc traveled through (part of the path), δs, and the angles ϕ_1, ϕ_2, at the beginning and end of this arc are given, then the

increments in x and y position coordinates, δx and δy can be obtained as:

$$\delta\phi = \phi_2 - \phi_1$$

$$\delta x = \frac{\delta s}{\delta\phi}\cdot(\sin(\phi_1) - \sin(\phi_2)) \qquad \delta y = \frac{\delta s}{\delta\phi}\cdot(\cos(\phi_2) - \cos(\phi_1)) \tag{2.5}$$

However, if this integration is carried out through software, it would require the main processor to be interrupted very often. To alleviate this load, a separate processor (8085) is used for the sole task of performing the computations necessary to continuously keep track of vehicle position. Every time a pulse is sent by the wheel encoders, the gyroscope is polled and a new position is computed. As can be seen from the above formulae, the task reduces to 3 subtractions. 2 multiplications (δs is preset) and 4 trigonometric computations. Instead of computing sine and cosine values, the hardware uses look up tables.

The block diagram of the ground navigator is shown in Fig. 2-9. The 8085 processor is contained in the block labeled COMP UNIT. The data inputs to this block come from the wheel encoders, the gyroscope and the main computer (corrections of position and bearing). The outputs are the computed (x,y) position.

As described in Sec. 2.1.3, it is necessary to keep track of the different ground positions at which beacons were spotted in one goniometer sweep. To achieve this, the COMP UNIT has one input from the signal that indicates that a beacon has been sighted and one input from the INDEX PULSE issued by the goniometer indicating the end of one complete rotation of the goniometer. When the INDEX PULSE is issued, the buffer BUFFER is initialized. Now, every time a beacon is sighted, the current vehicle position in COMP UNIT is stored in the buffer. This allows for the vehicle movement to be compensated

for in the LOCATE algorithm and also provides a rough estimate of vehicle position at any given point.

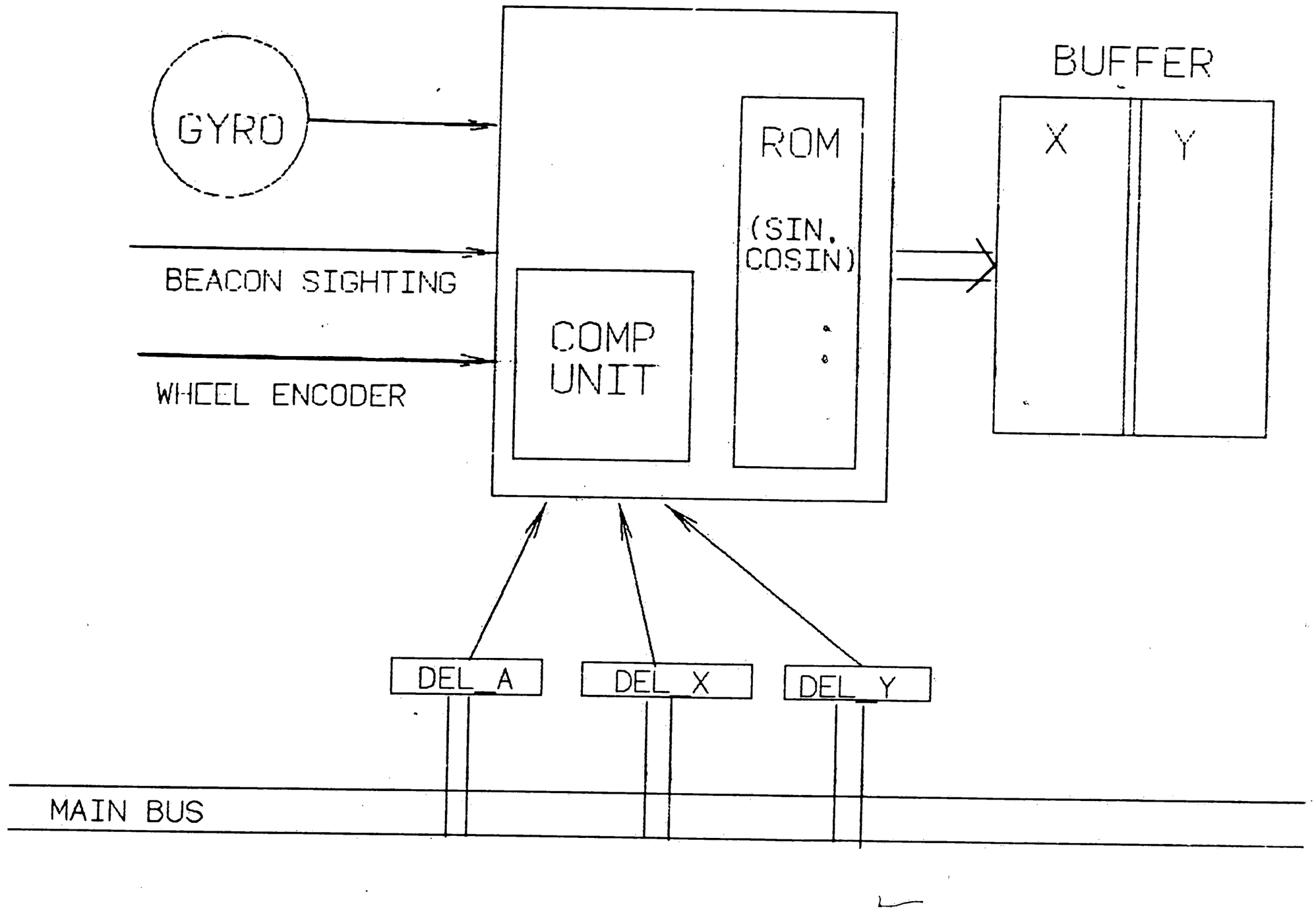


Figure 2-9: The Ground Navigator: Functional Block diagram

The main computer communicates with the Ground Navigator through the registers DEL_A, DEL_X, DEL_Y in Fig.2-9. This fulfills two needs:

- The gyro only provides a floating output. The main computer has to constantly keep track of what output corresponds to the zero angle. To start off with the vehicle is in a known position and orientation. For example, if the vehicle is oriented at 30 degrees and the 12 bit output of the gyro is the number 200 then the zero angle is at 115. The main computer has to inform the the ground navigator of this offset.

- Every time a "fix" is made, the computer is able to determine the error that is in the ground navigator (at the INDEX PULSE). The errors in position and orientation have to be communicated to the ground navigator and are done so through the registers mentioned.

2.2 Navigating the Specified Path

The vehicle used has analog servo systems through which speed and angular position of the front wheel can be specified. These servos are supplied with a voltage corresponding to the speed and angle required. There is a further capability of specifying the length of a path segment (distance to be traveled by the front wheel at the specified angle). After the specified segment has been traveled, a "ready" signal is returned to the computer which triggers computation of the next path segment.

2.2.1 Moving Between Two points

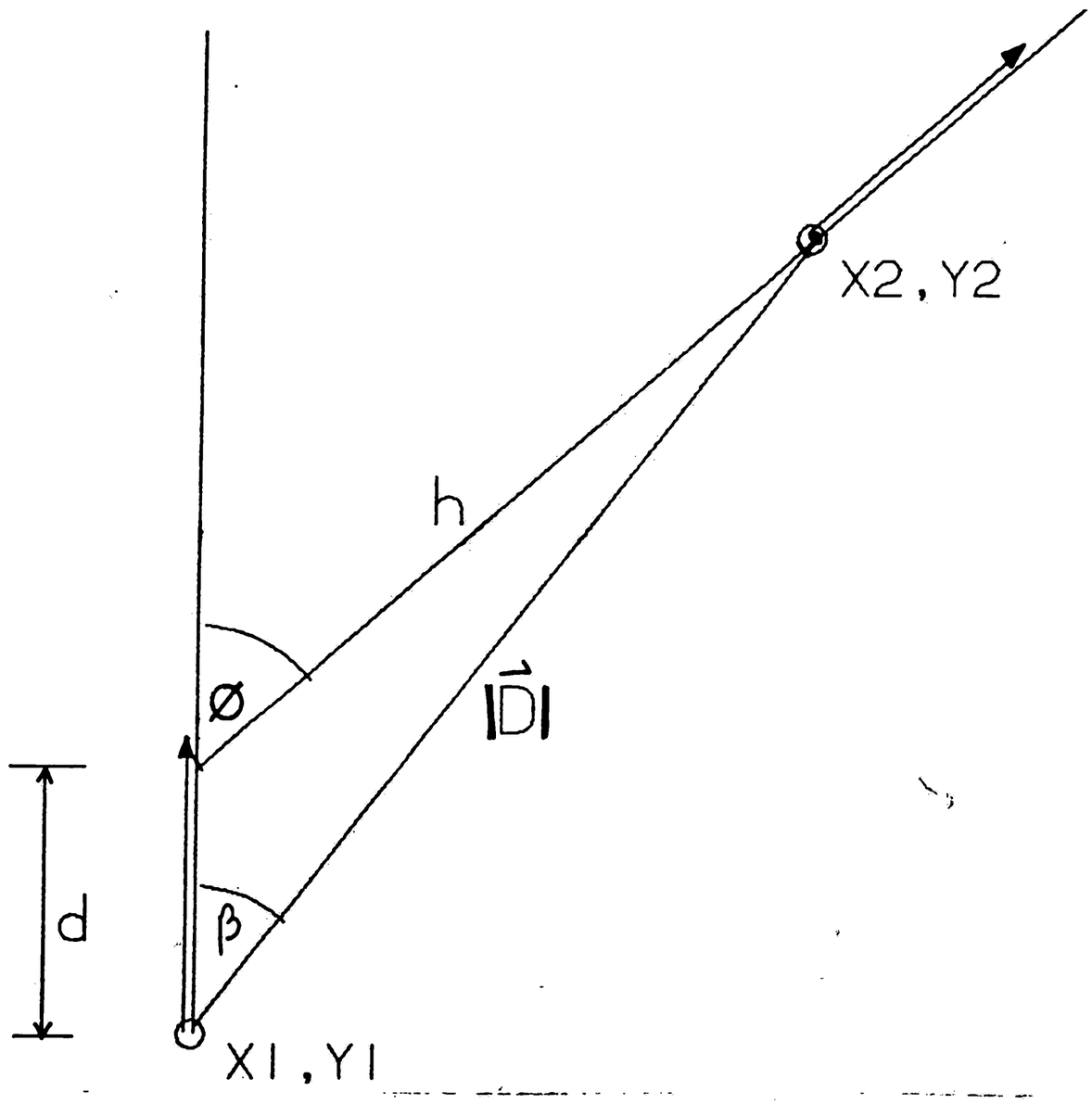


Figure 2-10: Moving between two points with arbitrary orientations

The ability to make the vehicle travel between two points with arbitrary

starting and ending orientations[2] is important for successful navigation. The problem of plotting a smooth path from $(X_1.Y_1)$ to $(X_2.Y_2)$ (See Fig. 2-10) can be subdivided into several cases depending upon the orientation angles α and β. Except for the case requiring the vehicle to move dead ahead without turning, every other case results in prescribing two arcs of specific length and radii of curvature. Note that a straight line is an arc with an infinite radius.

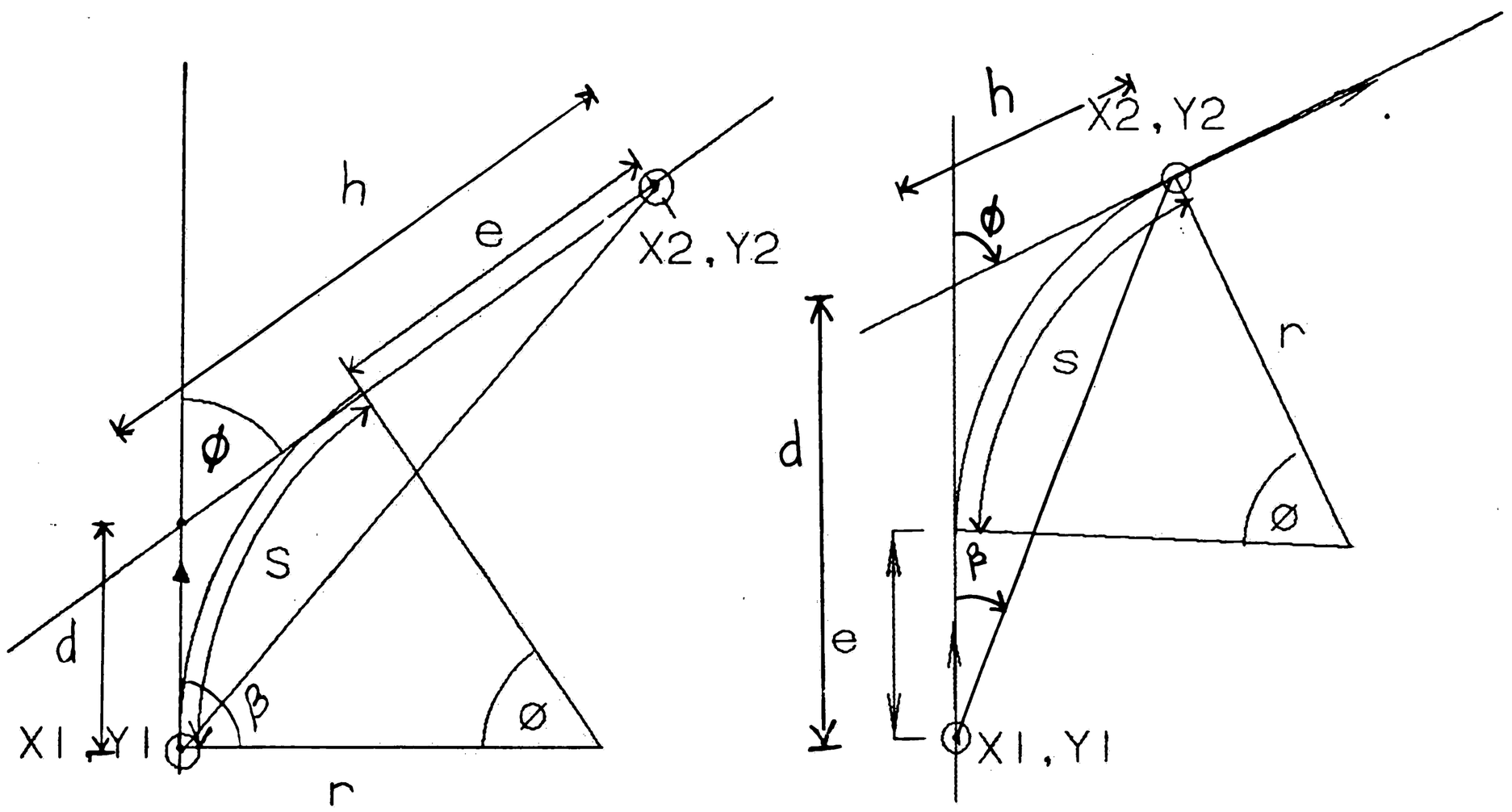


Figure 2-11: A smooth path using one arc and one straight line

In most cases, the robot can go from (X_1,Y_1) to (X_2,Y_2) in either of the two ways shown in Fig. 2-11.

- If $\beta < \phi < 2\beta$: first follow an arc an then a straight line.

---

[2]We assume the path is specified in such a manner that there is at most a 90 degree difference between the vehicle orientation required at two consecutive points.

- If $\phi > 2\beta$: first follow a straight line and then follow an arc.

Notice that the same applies for the mirror images of the cases shown in Fig. 2-11.

For $\beta < \phi < 2\beta$, r (the radius of curvature), s (arc length), and e (length of straight line) are computed in the following manner:

$$r = \frac{d}{tan(\phi/2)}$$

$$s = r \cdot \phi \qquad e = h - d \tag{2.6}$$

For $\phi > 2\ \beta$, the outputs r, e, s are:

$$r = \frac{h}{tan(\phi/2)}$$

$$e = d - h \qquad s = r \cdot \phi \tag{2.7}$$

The case where $\phi < \beta$ is special because it is not possible to get from **A** to **B** without using a 'S' shaped curve. Further, if the direction of translation is opposite to that of rotation, the path is necessarily S shaped irrespective of the angle ϕ. Although it is unlikely that the path specified to the mobile robot would require such manoeuvres, it is important to consider such and other cases that would require more than a simple arc motion for two reasons:

- Since the robot may be at any orientation before it starts moving, and similarly may be required to be any orientation when it stops, the only way to accommodate such situations would be to make a double turn (follow a S shaped path).

- Since there is no real time feedback built into the drive controller, and the fact that at this point, the dynamics of the steering mechanism has not been characterized, the actual path traced in response to a drive command might put the vehicle in such a situation that to get back on track it is necessary to take an S shaped path.

The 3 cases requiring S shaped paths may be classified as:

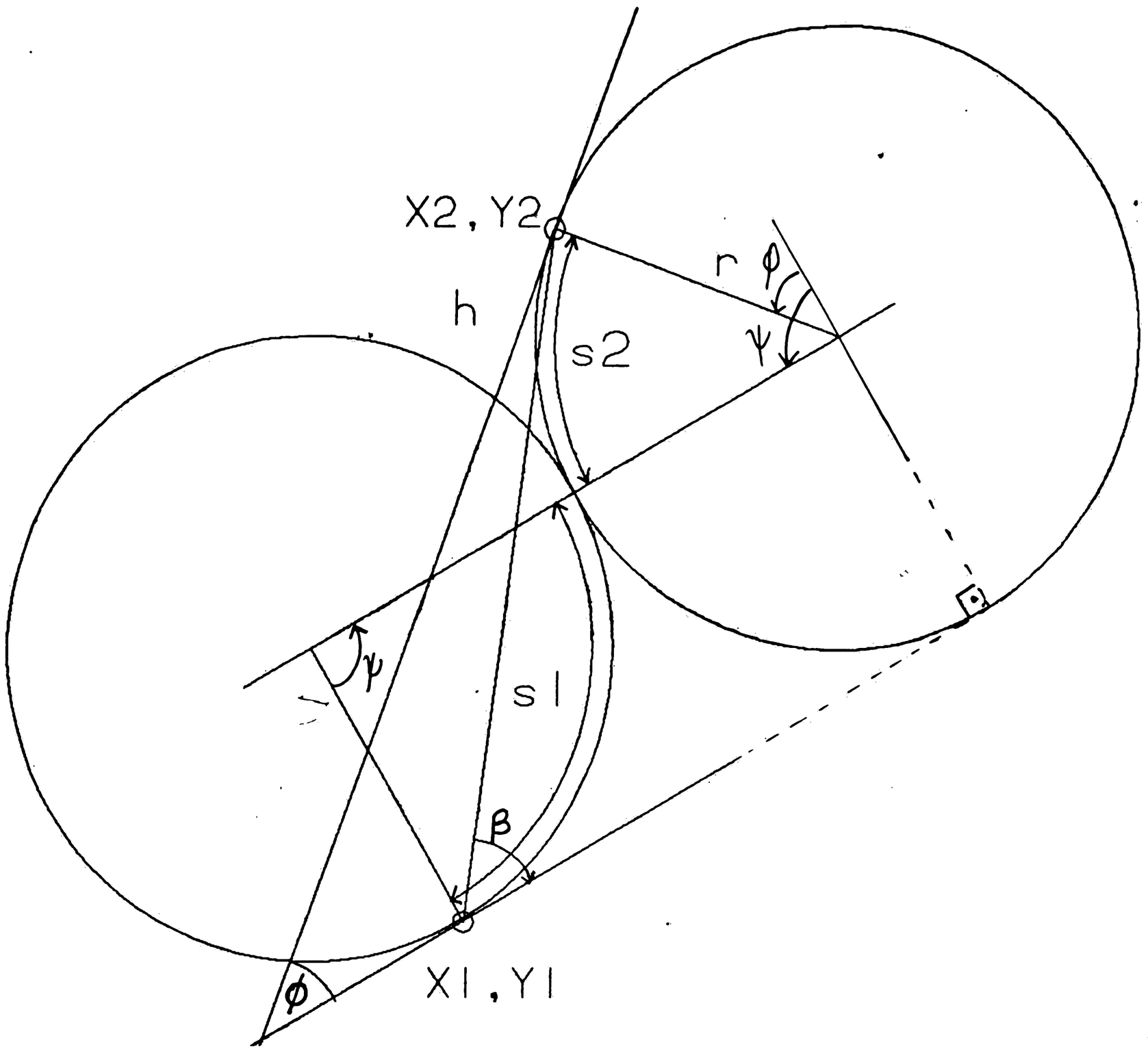


Figure 2-12: **S** shaped path when $\phi < \beta$

1. $\phi < \beta$, and translation and rotation are in the same direction (Fig. 2-12). (Note: Eq. (2.8) applies for $0 < \phi < \beta$) In this case the required arcs are described by:

$$\psi = \cos^{-1}\left[\frac{\cos(\phi) + \tan(\beta)\cdot\sin(\phi) + 1}{2\cdot\sqrt{1 + (\tan(\beta))^2}}\right] + \beta \tag{2.8}$$

$$r = \frac{a}{\cos\phi - 2\cdot\cos\psi + 1}$$

$$s_1 = r\cdot\psi \qquad s_2 = r\cdot(\psi - \phi)$$

2. The case where $\phi > \beta$ *and* the radius of curvature required is less than

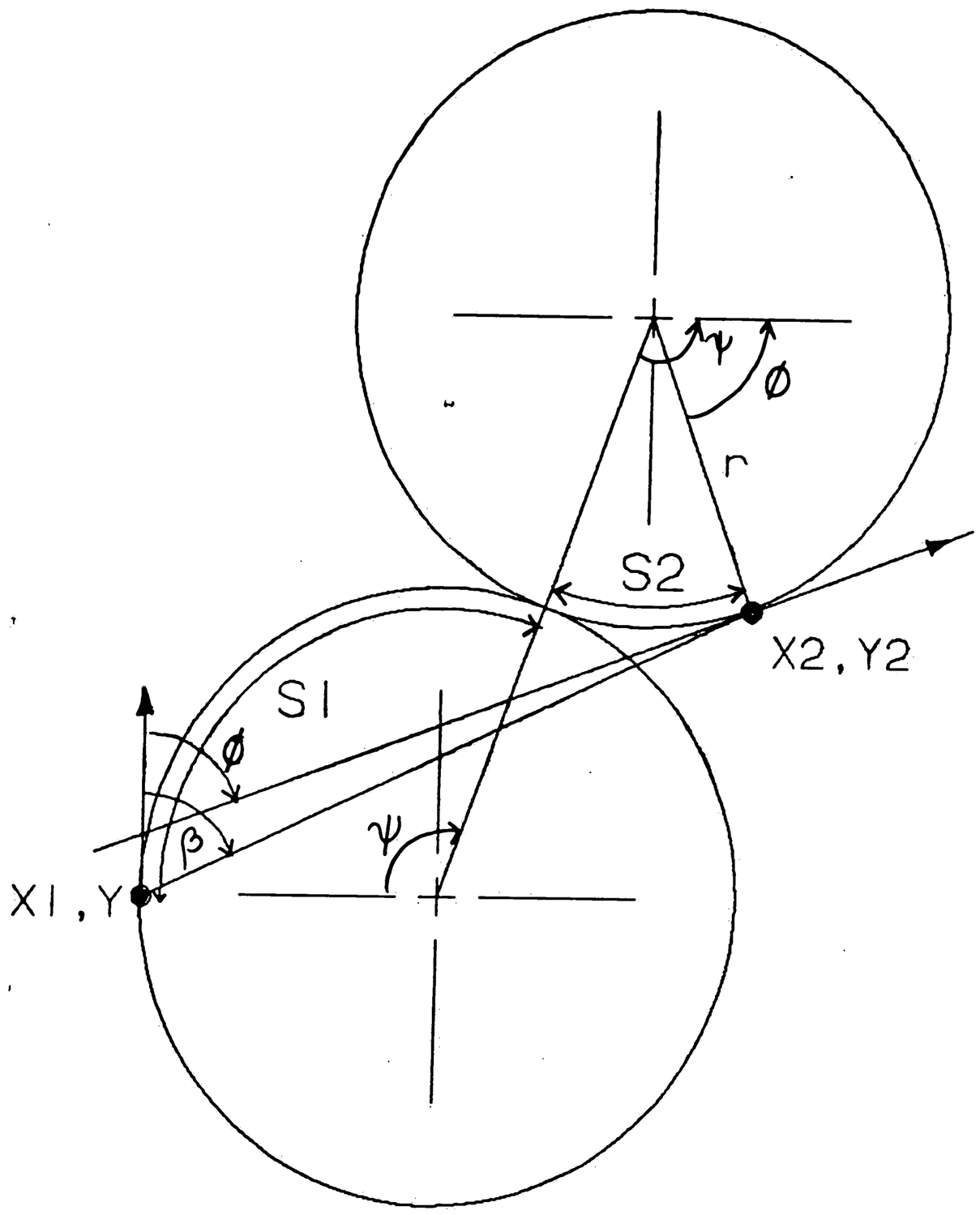


Figure 2-13: S shaped path for $\phi > \beta$, but radius < minimum radius

the minimum radius that the vehicle can turn around. Here too the rotation and translation are in the same direction (Fig. 2-13). For $\beta < \phi < 2\beta$ use Eq. (2.8). For $\phi > 2\beta$ use Eq. (2.9).

$$\psi = \cos^{-1}\left[\frac{\sin(\phi) + \tan(\beta)\cdot\cos(\phi) + 1}{2\cdot\sqrt{1 + (\tan(\beta))^2}}\right] - \beta \qquad (2.9)$$

$$r = \frac{a}{2\cdot\cos\psi - \sin\phi - 1}$$

$$s_1 = r\cdot\psi \qquad s_2 = r\cdot(\psi + \phi)$$

3. Translation of vehicle position is to the right but rotation of vehicle

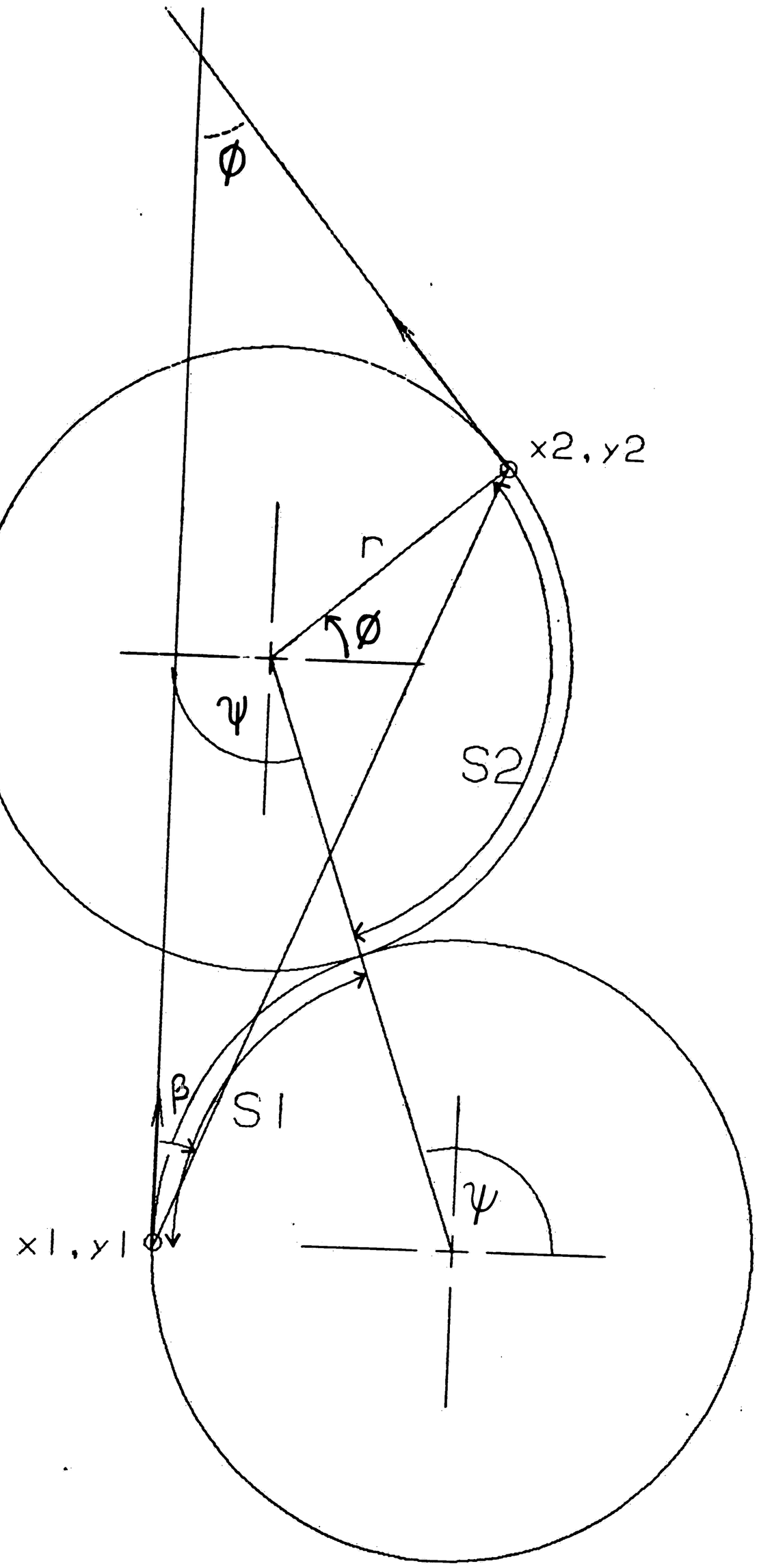


Figure 2-14: S shaped path when translation and rotation are opposite.

orientation is to the left *or* translation is to the left but rotation is to the right (Fig. 2-14). The arc description in this case turns out to be:

$$\psi = \cos^{-1}\left[\frac{\tan(\beta)\cdot\sin(\phi) - \cos(\phi) - 1}{2\cdot\sqrt{1 + (\tan(\beta))^2}}\right] - \beta \qquad (2.10)$$

$$r = \frac{a}{2\cdot\cos\psi - \cos\phi + 1}$$

$$s_1 = r\cdot(\pi - \psi) \qquad s_2 = r\cdot(\psi + \phi)$$

Note that each of the above classifications also apply to their mirror images. Also note that it even though it is possible to always use two arcs to travel between any pair of points, this method is used only when the path cannot be along a simple arc.

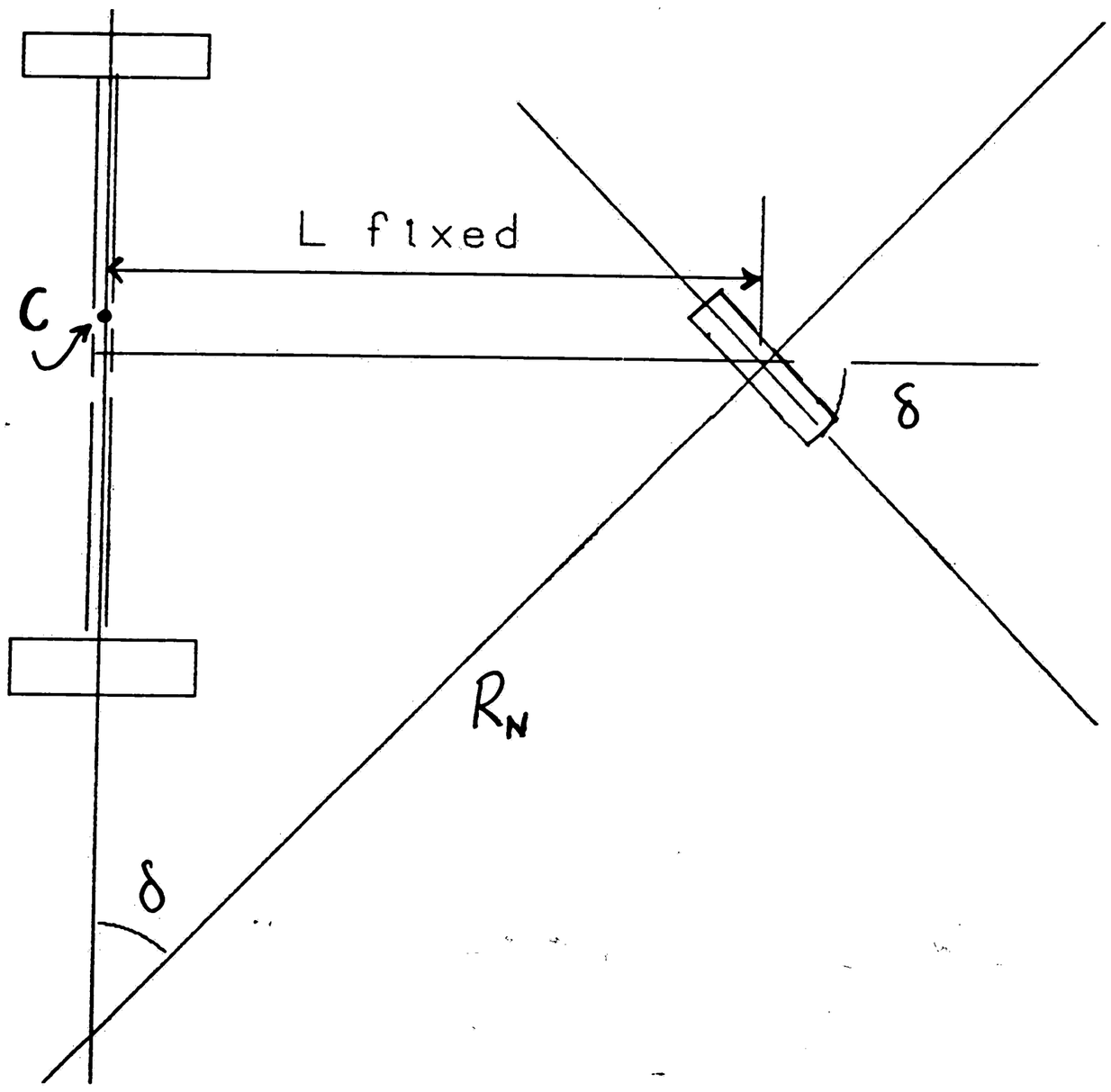


Figure 2-15: Calculating the Steering Angle

Because the steering controller requires a steering angle δ, an additional calculation is necessary as shown in Fig. 2-15. Note that the point marked **C** is where the goniometer is mounted. It is this point that represents the vehicle

when it is merely shown as a point. From Fig. 2-15 one gets

$$\delta = \tan^{-1} \frac{L_{fixed}}{R_N} \tag{2.11}$$

The steering controller accepts a voltage from 0 to 10 volts for the range of -90 degrees to +90 degrees. In our implementation, an 8 bit register is used to store the steering angle computed. Thus the number 0 stands for a -90 degree (right) turn and the number 255 stands for 90 degree (left) turn and hence the register is filled with:

$$\begin{aligned} &128 + \delta && \{\text{if left turn}\} \\ &128 - \delta && \{\text{if right turn}\} \end{aligned} \tag{2.12}$$

2.2.2 Planning Path Segments

Planning a path for a robot given only starting and ending orientation is dealt with in the next chapter. Presuming that such a path (a set of points joined by straight line segments) has already been found, the following is a description of a procedure that (using the methods developed in the previous sections) is responsible for making the robot follow this path. Fig. 2-16 shows a simple 4 point path with starting and ending orientations. If only the primary (specified) points are used with the equations given earlier, the path of the vehicle will include those points but will not track the straight line segments between them very closely. To trace the path with small error and tightest possible turns, more points are added to the path and some of the corner points are discarded. This list of secondary points which is actually used to plan the path is generated as follows: For long straight path segments, points are added so that the longest segment is twice the nominal radius (R_N). The nominal radius is the radius of curvature used by the vehicle while going around a

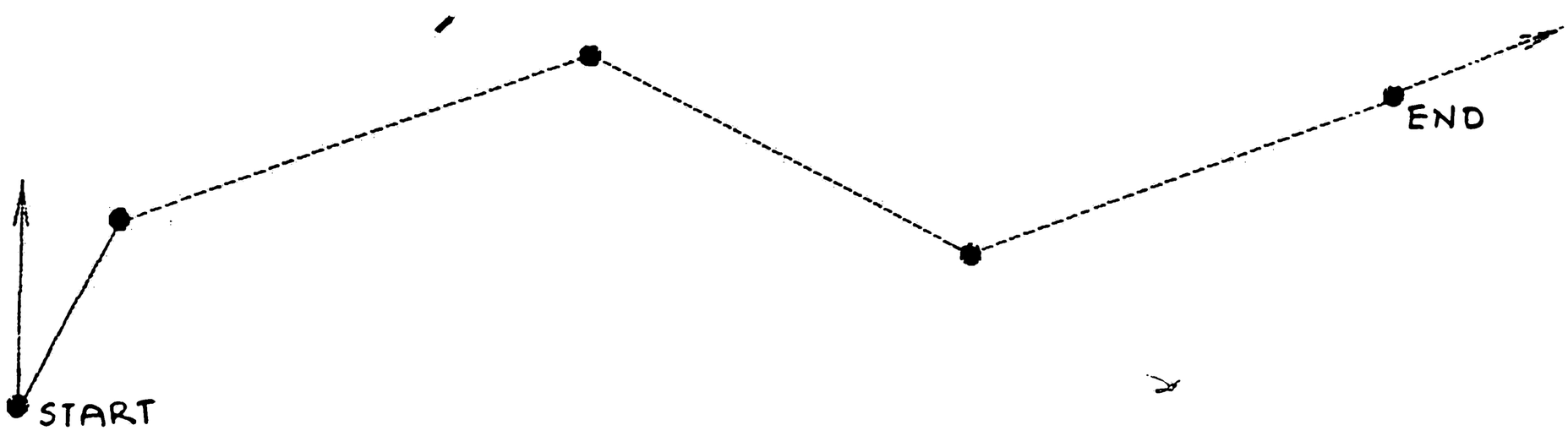


Figure 2-16: Initial Path Specified with Starting and Ending Orientations

corner and is function of load and speed of the vehicle. For a path that requires the vehicle to negotiate a corner at (X_i,Y_i), points are added at a distance of l_1 on either side of the corner and the corner point is discarded. l_1 is determined as

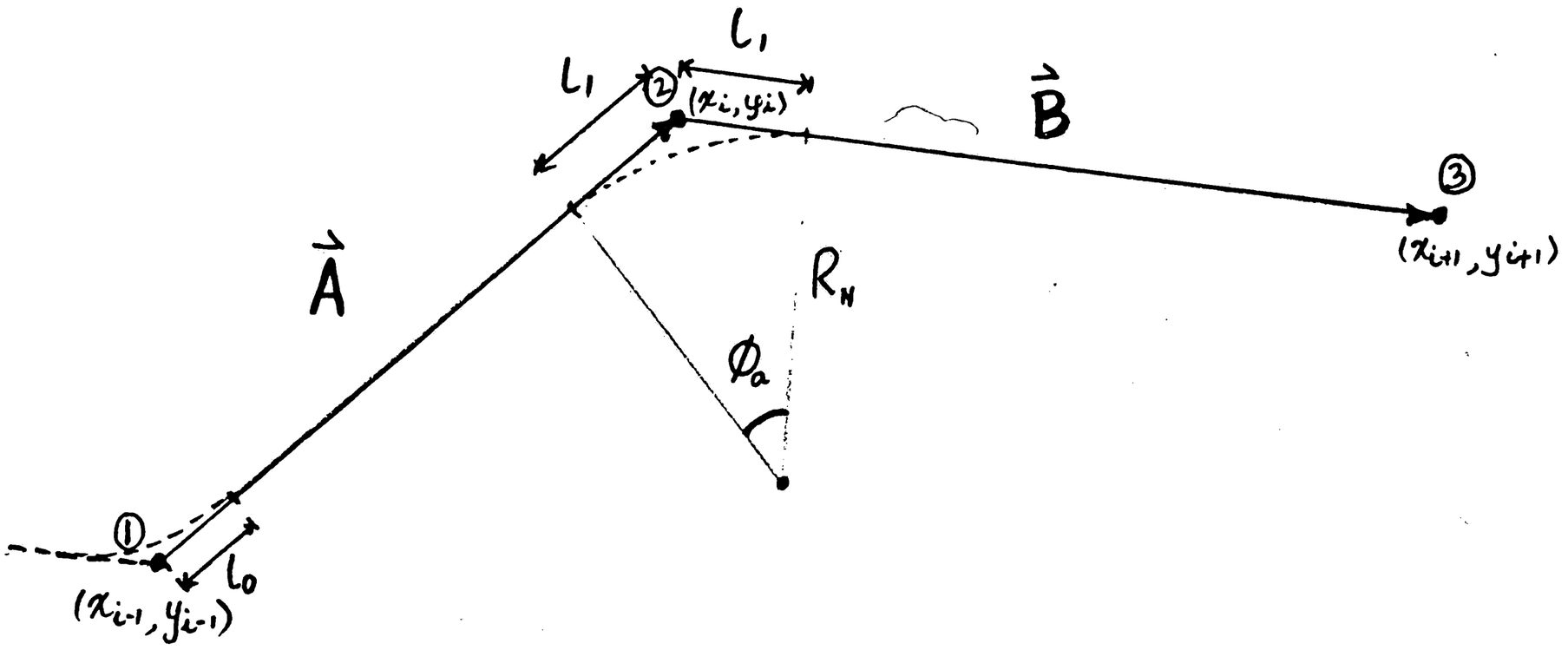


Figure 2-17: Adding Additional Points

$$\vec{A} = (X_i - X_{i-1}, Y_i - Y_{i-1})$$

$$\vec{B} = (X_{i+1} - X_i, Y_{i+1} - Y_i)$$

$$\phi_a = \sin^{-1} \frac{|(\vec{A} \times \vec{B})|}{|\vec{A}| \cdot |\vec{B}|} \qquad (2.13)$$

$$l_1 = R_N \cdot \tan \frac{\phi_a}{2}$$

l_0 is the distance between each secondary point and the previous corner. This process is shown in Fig. 2-17. If $l_0 + l_1 > |\vec{A}|$ then the corner point is retained and no extra points added.

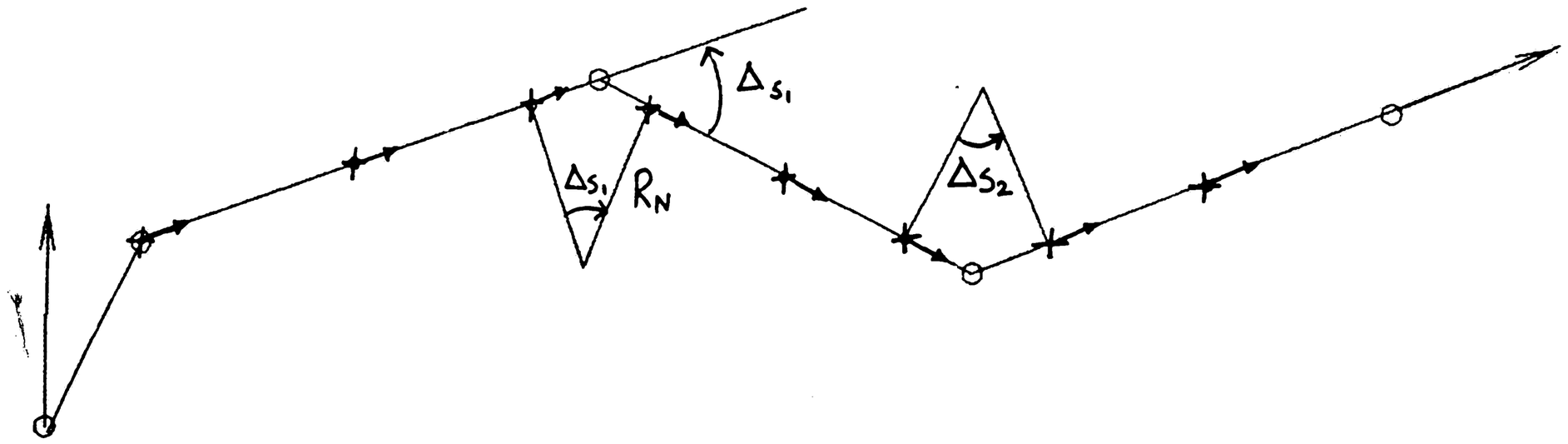


Figure 2-18: Secondary Points of the Path of Fig. 2-16

Fig. 2-18 shows the path of Fig. 2-16 superimposed with secondary points. Each of these points has a vector associated with it that stands for the direction of the vehicle at that point. The ideal path between any two consecutive points is predicted using the formulae presented in the previous section to determine the turning angles and path segment lengths. Fig. 2-19

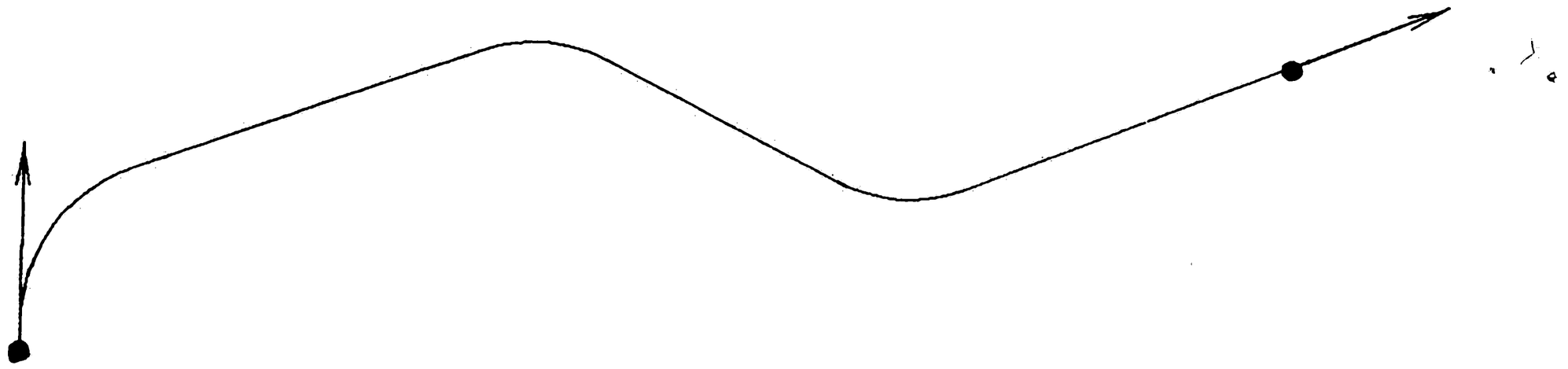


Figure 2-19: Ideal Vehicle Path for Points of Fig. 2-16

shows the ideal path taken for the points specified in Fig. 2-16.

2.2.3 Path Control Strategy

Normally, one would use real time control (perhaps through a dedicated processor) to keep the vehicle on a specified path. For the purposes of this experimental vehicle we have incorporated a pseudo-control method to compensate for factors such as vehicle dynamics (wheel slippage, sluggish steering response, etc.) which have not been quantitatively characterized and which force the vehicle away from the ideal path. To deal with such a problem, steering angle and path segment length are dynamically calculated at the end of each path segment so as to try to bring the vehicle back on track. Fig. 2-20 shows this process. The path from **A** to **C** is first computed as two arcs of lengths s_1 and s_2 passing through an intermediate point **B**. A short while before the vehicle has traversed the distance s_1 (δs before), the steering controller issues a pulse. At this time a position reading is taken from the

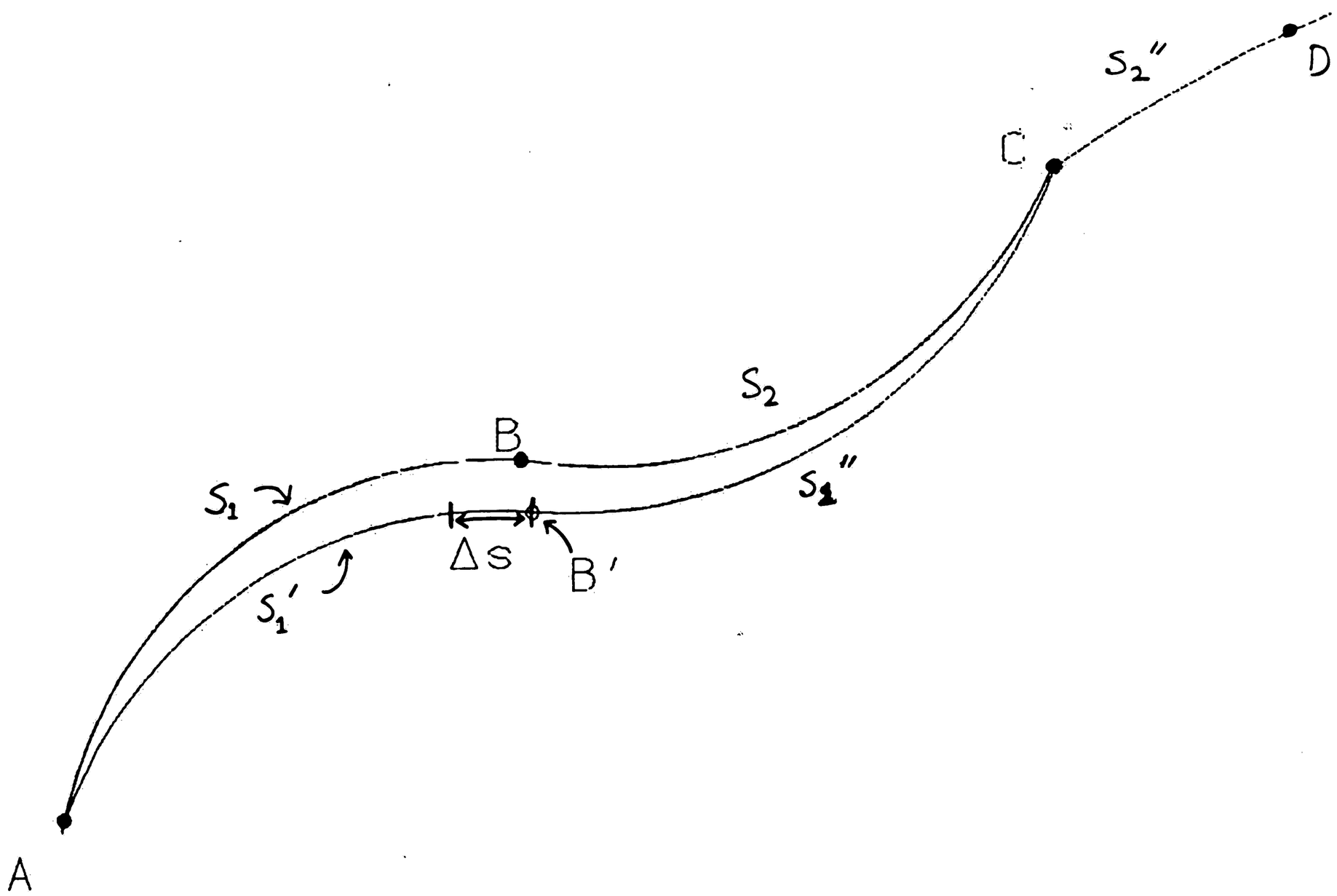


Figure 2-20: Compensating for Deviations from the Ideal Path

Ground Navigator. In Fig. 2-20, s_1' is the arc on which the vehicle actually travels, ending up at point **B'** instead of **B**. Now, 2 new arcs s_1'' and s_2'' are computed using points **B'** and **D**. Note that each case the second arc is never used (because at the end of each first arc, two new arcs are computed) except in the case that the first arc is very small. Then the arc used is half the length of the second arc at the end of which two new arcs are computed.

2.2.4 Tieing it All Together

Fig. 2-21 shows how the entire system is interfaced together. The part inside the dotted lines is implemented as software that is intended to run on the onboard computer (Intel 86/05). This computer is based upon the 8086 processor and will operate in conjunction with a 8087 math coprocessor. The other blocks represent hardware components.

The only inputs that need to be supplied to the configuration as shown are:

- **Path Points**: This is a set of points that the vehicle must go through. Only those points at which a turn is necessary need to be supplied. These points may either be explicitly specified or may be obtained from a program based upon the automatic path planning procedures described in the next chapter (in which case only starting and ending points and orientations need be specified).

- **Nominal Radius**: This is the default radius of curvature that the vehicle will use while going around a corner. This radius will depend on the load of the vehicle and the speed required.

- **Nominal Speed**: The default speed that the vehicle will travel at. For sharp corners and/or heavy loads, the vehicle speed might have to be changed.

The block marked PATH REFITTING is described in Sec. 2.2.2. This block generates a set of vectors corresponding to vehicle position and orientation at each of the points on the (secondary) path. The following block marked PATH CONTROL (Sec. 2.2.3) is responsible for sequencing pairs of vectors and supplying them to the next block marked DRIVER. The actual lengths of arcs and steering angle required is determined here (Sec. 2.2.1) and sent to the hardware drive controller which is responsible for the physical execution of driving the vehicle over a path specified by these parameters. Beacon angles have to be modified to compensate for vehicle movement as described in Sec.

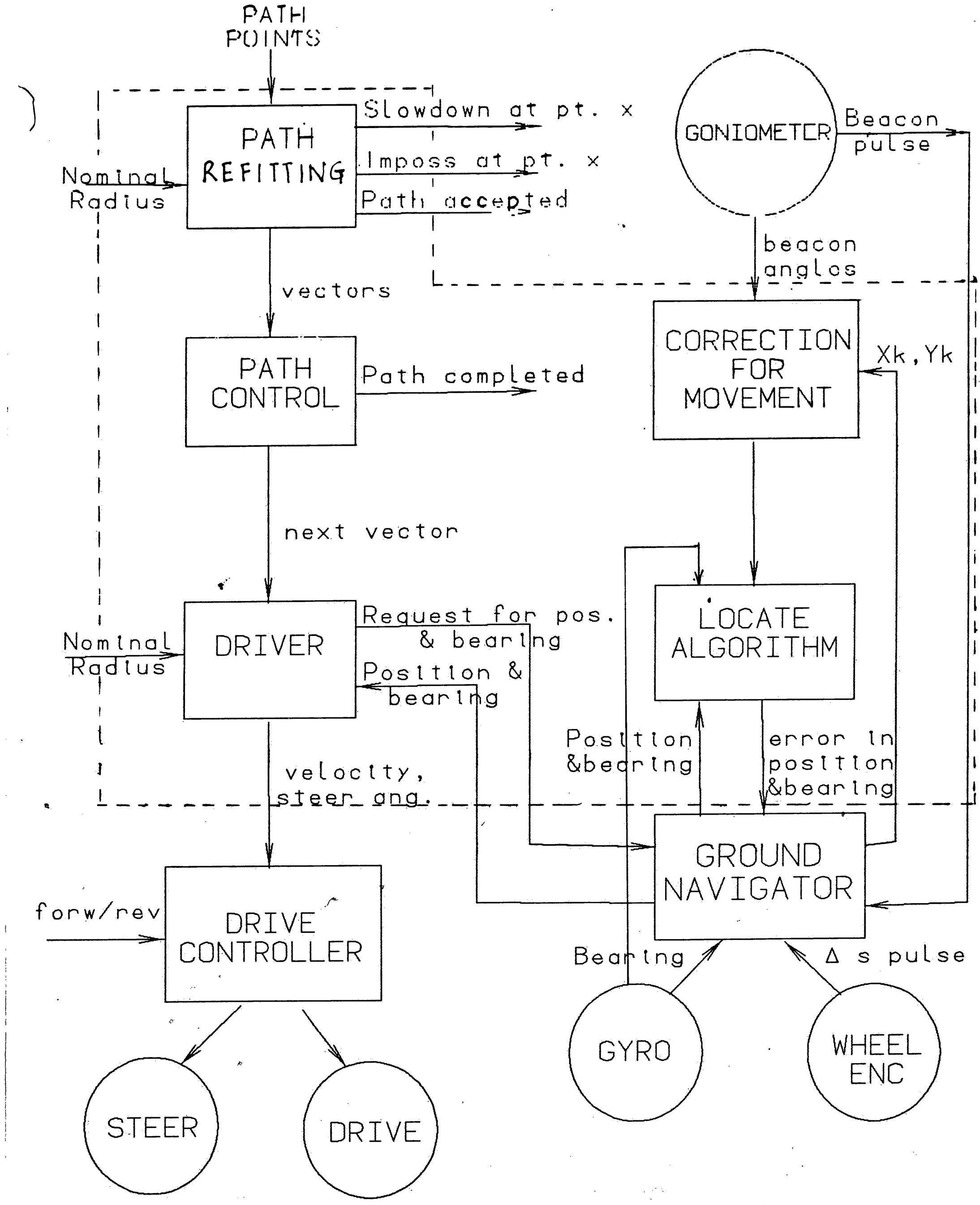


Figure 2-21: Functional Block Diagram of the Entire System

2.1.3 before the LOCATE algorithm can be used to obtain a "fix" (Sec. 2.1.1). The main computer then evaluates the error that has built up in the Ground Navigator (Sec. 2.1.5) and relays the error back to it. This periodic correction keeps the navigation system on course.

Chapter 3
Path Planning

3.1 Background

The previous chapter has dealt with algorithms to make a robot traverse a given path. The problem of actually generating paths that are both computable and optimal, turns out to be a non-trivial problem. Considerable attention has been focused on plotting paths between arbitrary source and destination points and a variety of different algorithms are now available to solve this problem [2, 3, 6, 7, 8]. Most of these algorithms however, yield either non-optimal solutions or are computationally expensive. A new path finding strategy that is computationally efficient and yields near-optimal results is presented here. It isolates and uses convex areas to find paths around obstacles.

The simplest approach to the find path problem for a mobile robot to traverse between workstations in a factory is to form a connectivity graph where each of the nodes represents a workstation and each link has associated with it a predetermined path of corresponding length. Then finding a path between workstations can either be done by searching the connectivity graph for the shortest path, or, looking up in a table of precomputed best paths from all possible source points to all possible destination points. However, this method is severely limited- there is no scope to start or to end at a point not included in the list of workstations. Also, modifying this list is time intensive as well as futile for the case where there may be arbitrary starting and ending points. This method has been used with some amount of success in the past in environments where the robots are only required to take a few paths many

times, with the probability of the paths changing being very low.

There has been significant research done to tackle this problem on a more general level and can be summarized by two approaches. Ignat'yev [6], and later, Lozano-Perez [9] used a technique called V-Graph which uses a graph of vertices between which travel is possible in a straight line. This is essentially a table of which nodes (vertices) are "visible" (can be traveled to in a straight line) from each node. This method, though similar to the one used by humans, when automated, has several drawbacks. Every time the source and destination points are specified, the graph has to be augmented with new nodes and new links. Thus, not only does the resultant graph have a large number of links but the establishment of these links is highly complex.

The other approach used more recently has been to partition the free space into convex polygons. The motivation for isolating convex shapes is the following: Any two points in a convex shape can be joined with a straight line without leaving the shape. If convex shapes can be found such that they represent areas free of obstacles, then a robot can travel between two points in that area without colliding into obstacles. Crowley and Chatila suggest breaking up the free area (for traversal) into non-overlapping convex shapes [10, 11]. Development of the path depends on traversing the connectivity graph that is produced by representing free convex polygons as nodes. Nodes corresponding to polygons with common edges are joined by arcs. The problem with a strategy that breaks up space into non-overlapping areas is that it fails to take full advantage of convexity and consequently misses some straight line paths that may belong to a convex area that the procedure is not aware of. This is a natural consequence of the fact that the need for non-

overlapping areas does overlook a considerable number of convex areas in the layout. Further, if the paths are not dynamically refitted to be optimal, paths that would be "naturally" straight, turn out to be quite contrived. This effect is particularly pronounced if there are relatively large free areas to contend with. However, there is a one-to-one correspondence between the source and destination points in free space and the graph nodes, and thus the method is successful in getting around the high computational expense at the cost of optimality.

Brooks [7] has proposed a method that hopefully combines the advantages of both the earlier procedures. Instead of determining the corners of objects that are visible, his method isolates free areas in the form of generalized cones. He also considers a finite sized rectangular robot. Most earlier methods shrunk the robot to a point and extended the size of the obstacles to be avoided. Brook's robot always traverses along the axes of free cones. and generously avoids the obstacles. The optimality is however lost because although the cones overlap, one does not make full use of convexity. Kuan, et al [12] further improved Brooks' method by using a mixed representation of free space. Their strategy used cones to represent narrow spaces, and non-overlapping convex polygons for larger free areas. Though their method works well for highly cluttered environments, the drawbacks associated with non-overlapping areas, still remain. Further, this modification does not reduce the graph complexity nor does it exploit any orderly orientation of the obstacles.

The path planning procedure outlined here derives further benefit from the concept of convexity by identifying *all* the largest rectangular free areas. In order to achieve near-optimality, without sacrificing computational efficiency, we

create a graph with nodes corresponding to each such convex area. Intersecting convex shapes are represented as adjacent nodes. Path planning is then reduced to finding a route from a source node to a destination node through the graph and choosing the best possible path based on a given cost function. In this work the cost function used is the length of the path. i.e we always choose the shortest possible path. In order to improve the computational complexity, and to provide a reasonable data base, we choose the obstacles as well as free areas to be rectangular in shape. The obstacles are grown in size and equivalently the robot is shrunk to a point to simplify path planning without collisions as in [9]. We use a back-tracking algorithm to avoid repetitive cost evaluation. Further, a dynamic cost allocation method allows for a rapid establishment of the optimal path.

An advantage of our method is that it allows the extraction of a near straight line path if one exists. The number of nodes and links in the graph are $\mathbf{O}(n)$ in most cases (where n is the number of rectangular obstacles). The graph complexity is reduced if there is an orderly orientation of obstacles. Further, the database and the graph generated are independent of particular source and destination points and can therefore be precomputed for a given robot environment.

3.2 Isolation of Prime Convex Areas

For simplicity, this work assumes that the areas that the robot cannot cross are rectangular with edges parallel to the coordinate axes. If either the workspace or the obstacles have nonrectangular edges, then rectangular edges are added. Edges have to be added in any case around all the objects to prevent the robot from getting too close to obstacles. This procedure, similar to that used in [8] is explained further in Sec. 3.5. Further, for computational simplicity we restrict our attention only to rectangular free areas (there are an infinite number of nonrectangular free areas). Given a map of the boundaries and the obstacles, the environment is partitioned by the edges of these shapes into a grid of at most $2n+1 \times 2n+1$ rectangles where n is the number of such inadmissible areas[3].

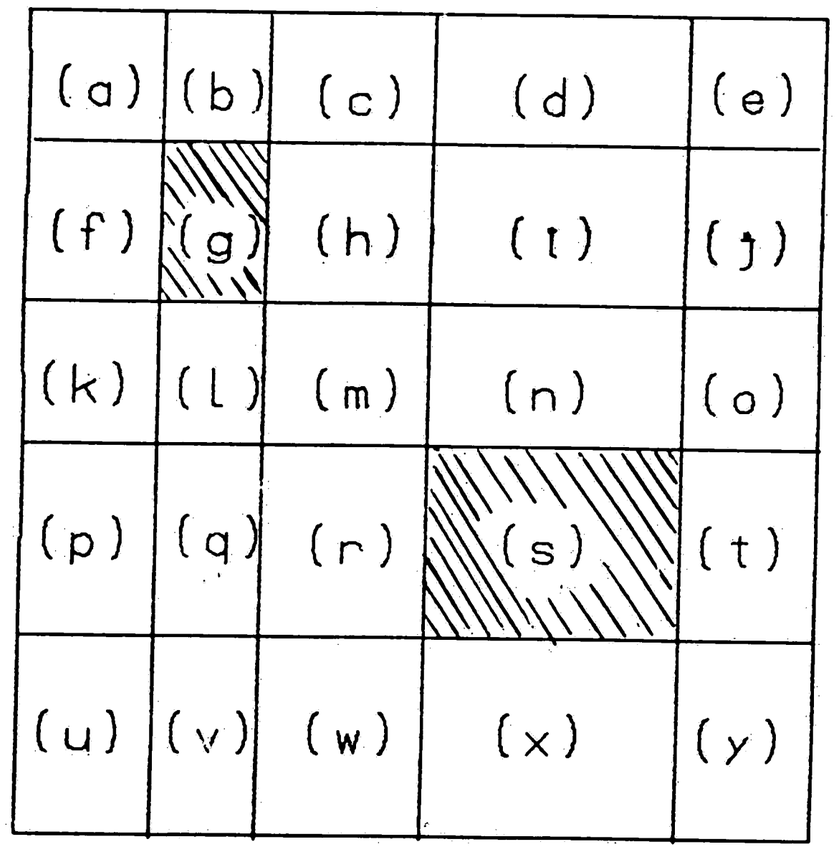


Figure 3-1: Partitioning the Environment

[3]This holds if none of the objects line up or overlap. Sec 3.5 describes other scenarios.

This makes the most sense if one notes that the convex areas we are after are bounded *only* by the edges of the obstacles. Each such rectangle can be represented by a pair of binary strings each at most $2n+1$ bit long. The left sub-string represents the relative x position and the right the y position. For example in the layout of Fig. 3-1, generated by the obstacles in cells g and s, a partition that is second from the left and third from the top (cell l), could be represented by the string

0 1 0 0 0 0 0 1 0 0

A similar notation can be used for areas made up of several of the rectangles. A string

1 1 0 0 0 0 0 1 1 0

for example, represents a convex area made up of 5 cells (k,l,p,q):

1 0 0 0 0 0 0 1 0 0,
0 1 0 0 0 0 0 1 0 0,
1 0 0 0 0 0 0 0 1 0 and
0 1 0 0 0 0 0 0 1 0.

A **prime convex area** is a free area which is not part of any other free area. Prime convex areas for a given layout are generated by fusing together the rectangular cells from the grid described above. Obviously, none of the cells marked as "inadmissible" by a robot can be used in such a combination.

Since we consider only rectangular convex shapes, we can use a procedure to "fuse" the individual partitions into prime convex shapes that is, in some sense, similar to the Quine-McCluskey technique [13] used to identify *all* the prime implicants of a logical expression. This procedure may be described as follows:

Step 1

Represent *each* horizontal strip (made up of horizontally aligned cells) by means of a pair of $2n+1$ length binary strings. Note that the right sub-string

will have only one bit set corresponding to the vertical position of the strip. The left sub-string has those bits set which correspond to the free rectangles in the strip. For example, the fourth horizontal strip of Fig. 3-1 has the representation

1 1 1 0 1 0 0 0 1 0

Step 2

Find all the contiguous horizontal strips. This is done by breaking up the left sub-string into contiguous runs of 1's and repeating the right sub-string in each part. For example, the strip of step 1 can be broken into two horizontal strips which represent contiguous free cells:

1 1 1 0 0 0 0 0 1 0 and

0 0 0 0 1 0 0 0 1 0

Step 3

Make a list of all strings generated by step 2 such that

1. Strings are grouped by identical right sub-strings.
2. Groups are ordered by the position of 1s in the right substrings.

Step 4

Generate a new list of strings from the old list of strings based upon the following rules:

1. The new i-th group of strings is generated by combining each string from the old i-th group with each string from the old i+1-th group. i = 1,2,...
2. Two strings are combined by (logically) OR-ing the right sub-strings and (logically) AND-ing the left sub-strings. If the new string has a null (all zero) left sub-string, discard that string. Otherwise, add it to the new list.
3. Every time a string is added to the new list, check off *all* the strings from the old lists that are covered by the new addition. A string S_1

is said to be covered by a string S_2 if logical OR-ing of the two strings yields S_2.

Step 5

Repeat step 4 if the new list generated has two or more groups.

Step 6

A string from any list that is not checked off represents a prime convex area for the layout.

It is easy to prove that the algorithm described above does indeed provide **all** the rectangular **prime convex areas**. When applied to the layout of Fig. 3-1, the lists of strings generated and the prime convex areas obtained are shown in Table 3-1.

List 1

1 1 1 1 1	1 0 0 0 0	(A)
1 0 0 0 0	0 1 0 0 0	
0 0 1 1 1	0 1 0 0 0	
1 1 1 1 1	0 0 1 0 0	(B)
1 1 1 0 0	0 0 0 1 0	
0 0 0 0 1	0 0 0 1 0	
1 1 1 1 1	0 0 0 0 1	(C)

List 2

1 0 0 0 0	1 1 0 0 0
0 0 1 1 1	1 1 0 0 0
1 0 0 0 0	0 1 1 0 0
0 0 1 1 1	0 1 1 0 0
1 1 1 0 0	0 0 1 1 0
0 0 0 0 1	0 0 1 1 0
1 1 1 0 0	0 0 0 1 1
0 0 0 0 1	0 0 0 1 1

List 3

1 0 0 0 0	1 1 1 0 0	
0 0 1 1 1	1 1 1 0 0	(D)
1 0 0 0 0	0 1 1 1 0	
0 0 1 0 0	0 1 1 1 0	
0 0 0 0 1	0 1 1 1 0	
1 1 1 0 0	0 0 1 1 1	(E)
0 0 0 0 1	0 0 1 1 1	

List 4

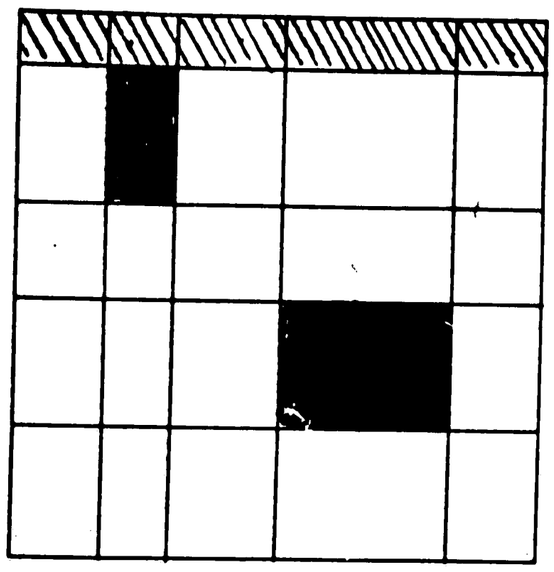
1 0 0 0 0	1 1 1 1 0
0 0 1 0 0	1 1 1 1 0
0 0 0 0 1	1 1 1 1 0
1 0 0 0 0	0 1 1 1 1
0 0 1 0 0	0 1 1 1 1
0 0 0 0 1	0 1 1 1 1

List 5

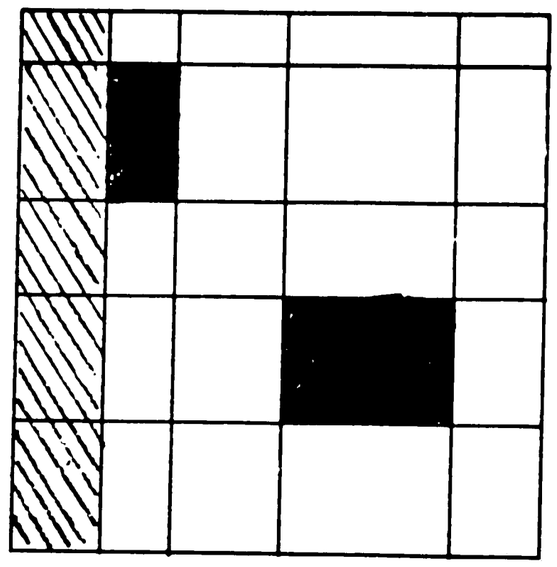
1 0 0 0 0	1 1 1 1 1	(F)
0 0 1 0 0	1 1 1 1 1	(G)
0 0 0 0 1	1 1 1 1 1	(H)

Table 3-1: Obtaining all the prime convex areas shown in Fig. 3-2 for the layout of Fig. 3-1.

Note: Prime convex areas remain unchecked and are labeled as (A) through (H).

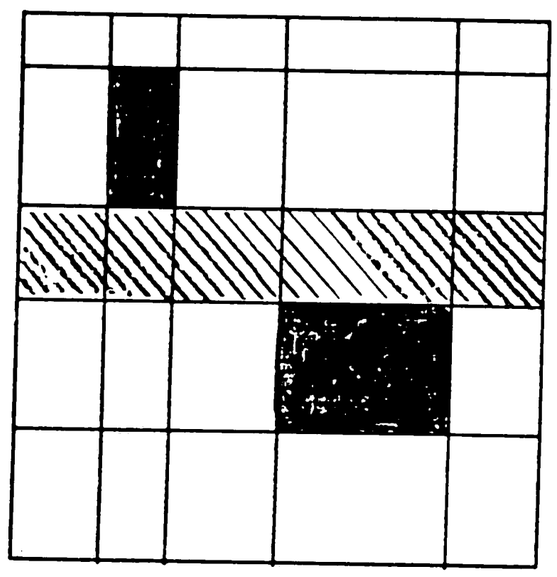


(A)



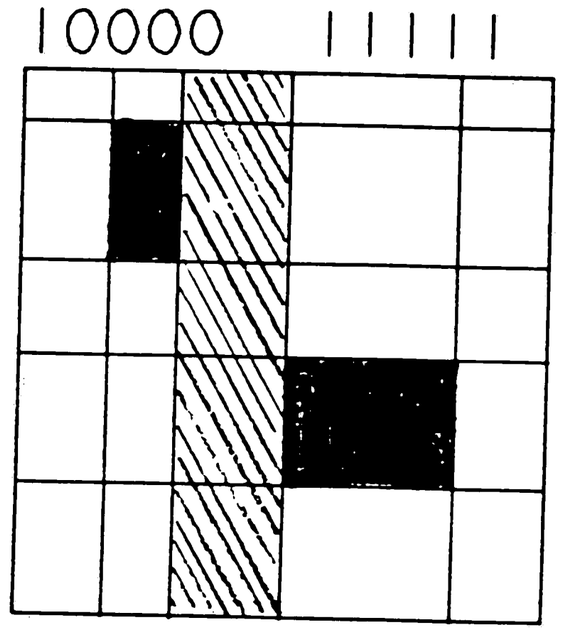
(F)

11111 10000



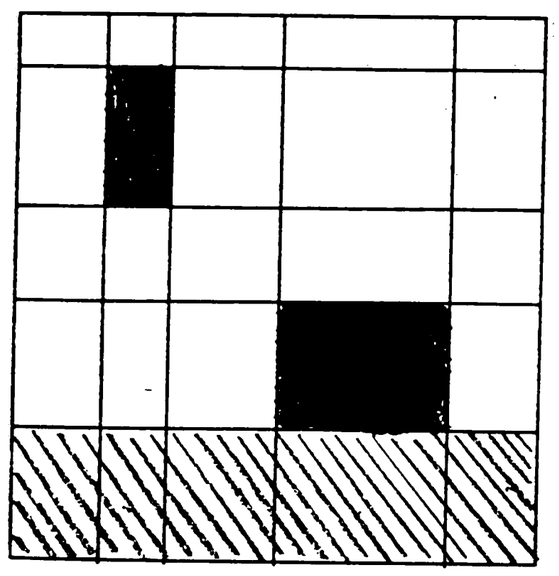
(B)

10000 11111



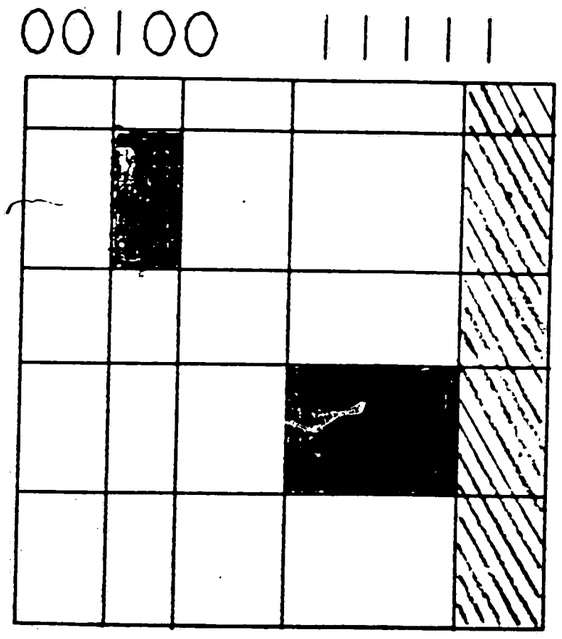
(G)

11111 00100



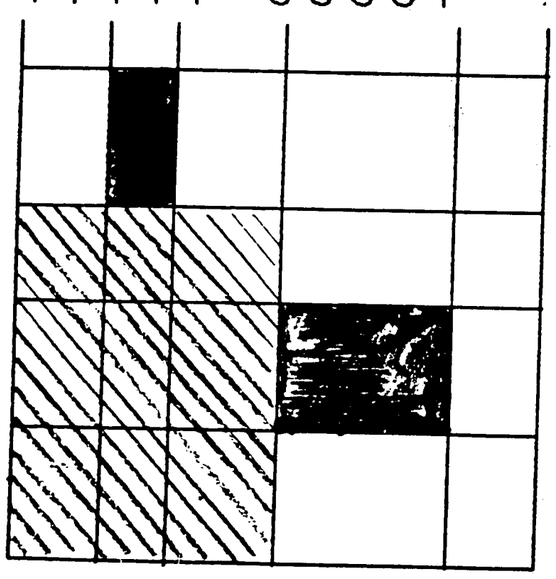
(C)

00100 11111



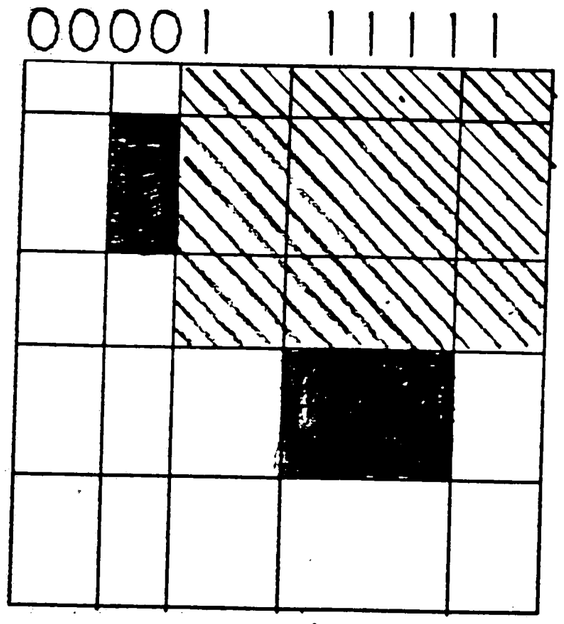
(H)

11111 00001



(E)

00001 11111



(D)

11100 00111

00111 11100

Figure 3-2: Convex areas in an environment containing 2 obstacles

3.3 Setting Up the Graph

The next step in path planning is the representation of information about the prime areas (generated in Sec. 3.2) in a convenient data structure. In order to facilitate the application of techniques such as orderly graph traversal and back-tracking, a graph is set up with prime convex areas as nodes. Two nodes are joined by an arc if the areas they represent, intersect. An arc has associated with it information about the area of intersection of the prime areas representing its ends. Fig. 3-3 shows such a graph obtained from the layout of Fig. 3-1.

If optimality is not a criterion, traversing the graph is straight forward. The prime convex areas in which the source and destination points are located may be determined, and the graph may be traversed from the source node to the destination node using one of a variety of techniques available [14, 15].

The consideration of optimality, however, brings about two complications:

- Both the source and destination points may fall inside several different nodes (since we have intersecting prime convex areas). Thus all possible paths originating from *valid* starting nodes and terminating on *valid* ending nodes have to be considered.

- Arcs cannot have fixed weights attached to them because any two points in one convex area are not necessarily equidistant to a point in another convex area. This means that the cost of traveling from one node to another is dependent on where the points are actually located in the convex areas and has to be computed every time a path segment is chosen.

The next section describes a strategy to overcome these difficulties and to choose an optimal path. However, it should be noted that the computation up to this point (i.e isolation of prime convex areas and setup of graph) needs to be done only once for a given robot environment.

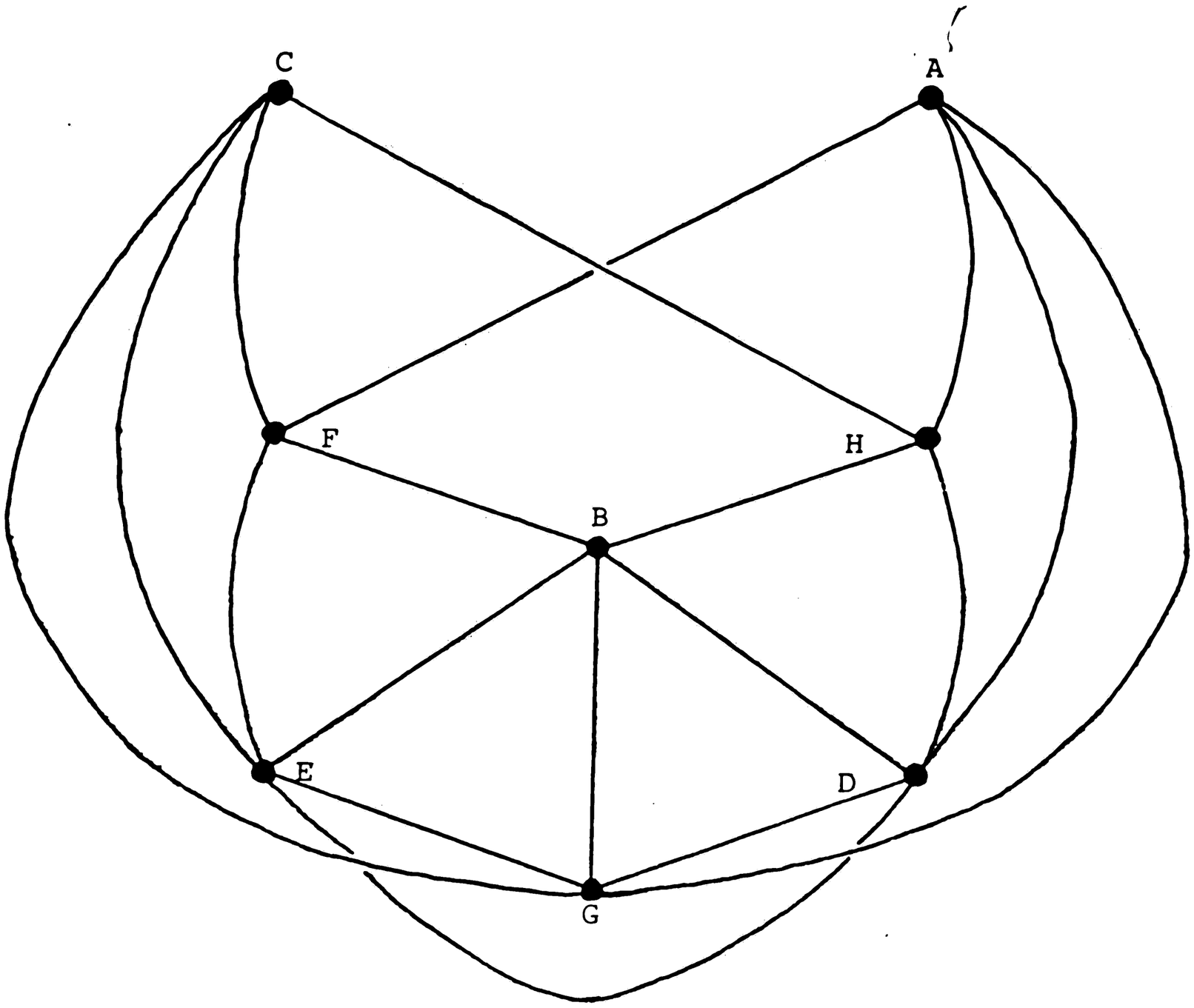


Figure 3-3: Graph of intersecting prime convex areas for the layout of Fig. 3-1

arc	between nodes	area of intersection
a	A - F	1 0 0 0 0 1 0 0 0 0
b	A - G	0 0 1 0 0 1 0 0 0 0
c	A - H	0 0 0 0 1 1 0 0 0 0
d	B - F	1 0 0 0 0 0 0 1 0 0
e	B - G	0 0 1 0 0 0 0 1 0 0
f	B - H	0 0 0 0 1 0 0 1 0 0
g	A - D	0 0 1 1 1 1 0 0 0 0
h	B - D	0 0 1 1 1 0 0 1 0 0
i	G - D	0 0 1 0 0 0 0 1 1 1
j	H - D	0 0 0 0 1 0 0 1 1 1
k	B - E	1 1 1 0 0 0 0 1 0 0
l	C - E	1 1 1 0 0 0 0 0 0 1
m	F - E	1 0 0 0 0 0 0 1 1 1
n	G - E	0 0 1 0 0 0 0 1 1 1
o	D - E	0 0 1 0 0 0 0 1 0 0
p	C - F	1 0 0 0 0 0 0 0 0 1
q	C - G	0 0 1 0 0 0 0 0 0 1
r	C - H	0 0 0 0 1 0 0 0 0 1

Table 3-2: Representation of the areas of intersection associated with each arc in Fig. 3-3

3.4 Dynamic Path Planning

The basic strategy of path planning involves traversing the graph generated in Sec. 3.3 from the node containing the source point to the node containing the destination point. Moving from a node to the next involves picking a point in the intersection of the two areas. However, as explained earlier, the choice of such a point affects the cost function associated with the travel and therefore merits further consideration.

Since the graph arcs cannot have fixed weights attached to them, the cost function must dynamically allocate costs to path segments as the path develops. Fig. 3-4 illustrates the graph traversal from node X_i to X_{i+1} and then to either node X_{i+2} or to X'_{i+2}. Let a, b and b' denote areas of intersection of X_i and X_{i+1}; X_{i+1} and X_{i+2}; and X_{i+1} and X'_{i+2}. Also denote by $mid(b)$ and $mid(b')$ midpoints of the two intersections. Let us assume that the current path has progressed till a point C_i in node X_i. For the sake of choosing the next segment of the shortest path, one should look ahead to nodes to be visited in the future. In the procedure presented here, we look ahead only one node[4]. The path segments are then computed as follows.

Assuming that the graph traversal is $X_i \rightarrow X_{i+1} \rightarrow X_{i+2}$, join points C_i and $mid(b)$ by a straight line. If the line intersects area a, then the next path point chosen is the point where the line first meets a. On the other hand, if the line does not intersect a, then the next point chosen on the path is the corner of a that is closest to the line. This second case is illustrated by the graph traversal $X_i \rightarrow X_{i+1} \rightarrow X'_{i+2}$. The point chosen is labeled C_{i+1} and the

---

[4]Looking ahead further than one node is computationally expensive and provides only marginal benefits Also, the more cluttered the environment gets, the less beneficial it is to look ahead by more than one node.

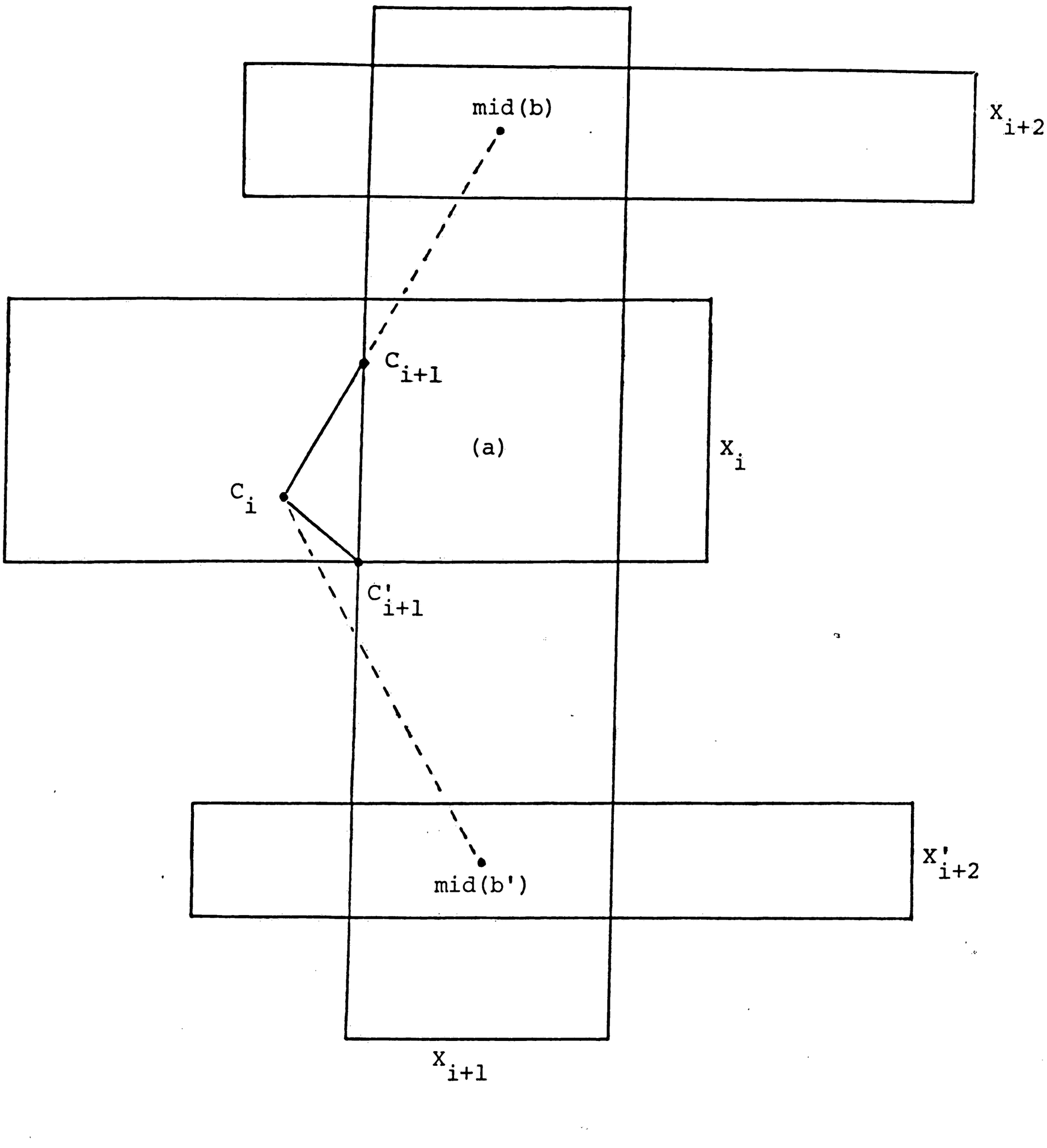


Figure 3-4: Development of a path segment based upon relative position of a future node

path segment $C_i \rightarrow C_{i+1}$ is added to the current path. For the purposes of this thesis, the cost allocated to this segment is merely proportional to its length. Since the new point C_{i+1} is now in X_{i+1}, a similar procedure could be used to continue path planning till the destination node is reached.

It can be seen that the cost allocation to the developing path lags behind the graph traversal by one step. Consequently, cost assignments cannot begin until the graph node path progresses at least to the third node. Similarly, when X_{i+1} is the final destination node (and therefore there is no X_{i+2} node), the final destination point itself is used in place of *mid*(*b*) in the above description to compute the last two path segments. The graph traversal technique used maintains two concurrent paths, one going through the graph nodes, X_i. and the other going through path points, C_i. Note that the list of C_i lags behind the list of X_i for the reason described above. Assuming X_i is the current node, C_{i-1} is the current point on the path and, **S** and **D** denote the sets of starting and ending nodes, respectively, the procedure reads as follows:

findnewnode[X_{i+1}]
    Choose X_{i+1} such that
        an arc exists between X_i and X_{i+1}
  and X_{i+1} is not on current node path
  and X_{i+1} is not in **S**
  and the arc between X_i and X_{i+1} is not marked {the arc has
                not already been considered and rejected}

moveforward[to X_{i+1}]
    Add X_{i+1} to current node path
    Determine C_i {as explained earlier}
    Add C_i to current point path
    Add cost of segment $C_{i-1} \rightarrow C_i$ to current cost
    i := i + 1

backtrack[from X_{i+1} to X_i]
    Unmark all arcs originating from X_{i+1}

```
        Mark arc from X_i to X_{i+1}
        Reduce the current cost by the cost of C_{i-1} → C_i
        Remove X_{i+1} from current node path
        Remove C_i from current point path
        i := i - 1

path planning {main program}
    determine S, D; bestcost := ∞ , currentcost := 0
    if S ∩ D ≠ nil then
        compute straight line path
    else
        for every X_0 ∈ S do
            i := 0; backtrackflag := false
            findnewnode[X_{i+1}]

            while (newnode exists) or (i > 0) do
                if backtrackflag = true then
                    backtrack[from X_{i+1} to X_i]
                    findnewnode[X_{i+1}]
                endif

                if newnode exists then
                    moveforward
                    if currentcost > bestcost then
                        backtrackflag := true
                    else
                        if X_{i+1} ∈ D then
                            copy best path
                            backtrackflag := true
                        else
                            findnewnode[X_{i+1}]
                        endif
                    endif
                endif
            endwhile
```

Results obtained by application of this procedure to some typical layouts are presented in the next chapter.

3.5 Other Considerations

The obstacles in the layout must be grown by at least half the longest side of the vehicle so that the vehicle doesn't collide into objects as it passes them. Fig. 3-5 shows the extension of non rectangular boundaries and obstacles. Depending on obliqueness of the edges, smaller or larger rectangles may be used to extend the obstacles. Fig. 3-6 shows the grid of cells created

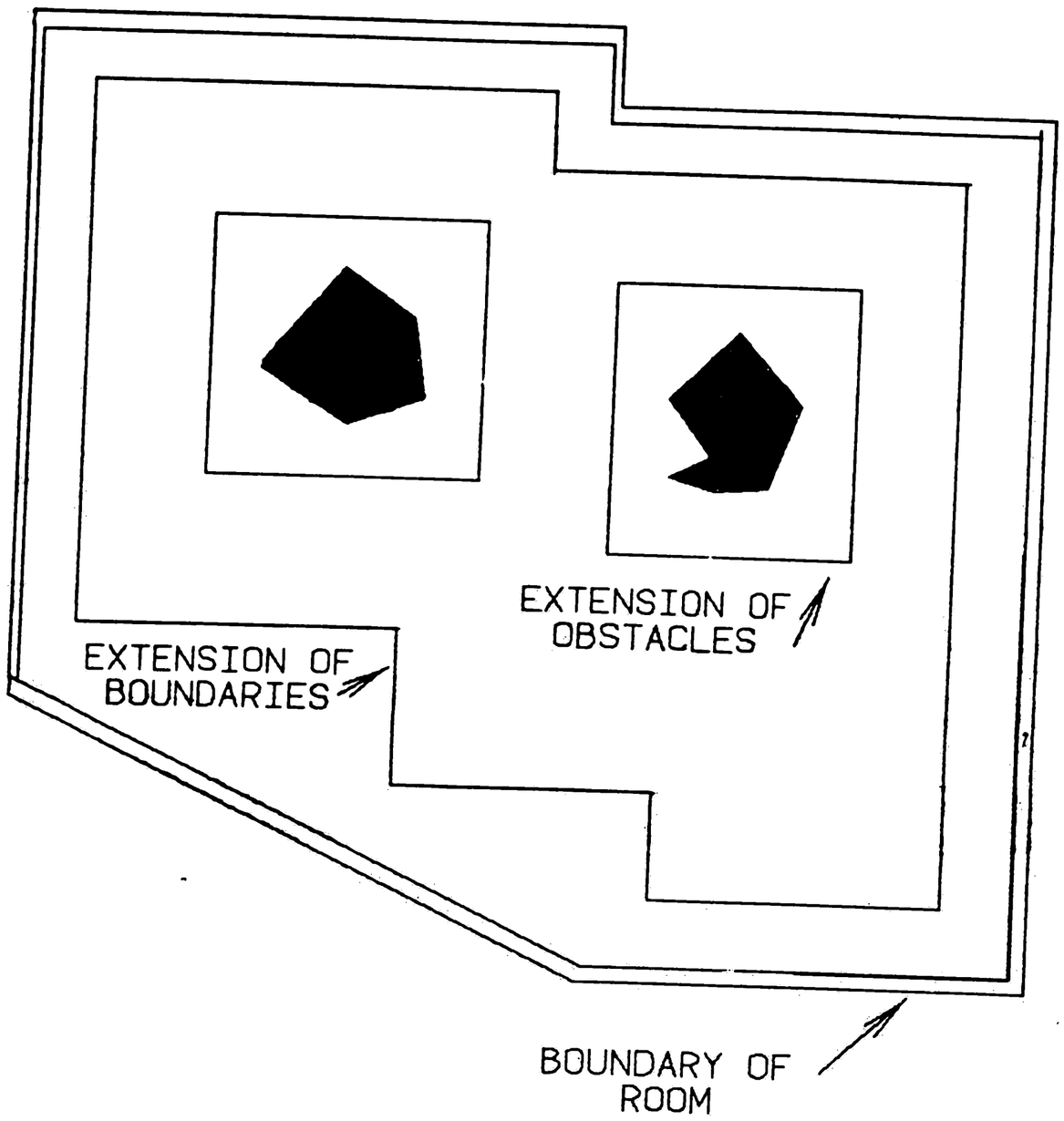


Figure 3-5: Growing Obstacles and Boundaries

by the layout of Fig. 3-5. Note that those cells that are "inadmissible" are crossed out.

Also note that "optimality" of paths is based upon the assumption that the robot is circular in shape. If the vehicle is non-rectangular, *and* the obstacles are so close that the widest part of the vehicle will not be able to get between them (but the narrowest part will), then the method described here will not find optimal paths. The vehicle described in Chapter 2, "Cyclopian" is a three wheeled rectangular vehicle and thus this problem will carry over to it also. It is assumed for the purposes of this work that the obstacles will be far enough apart that this will not be a consideration.

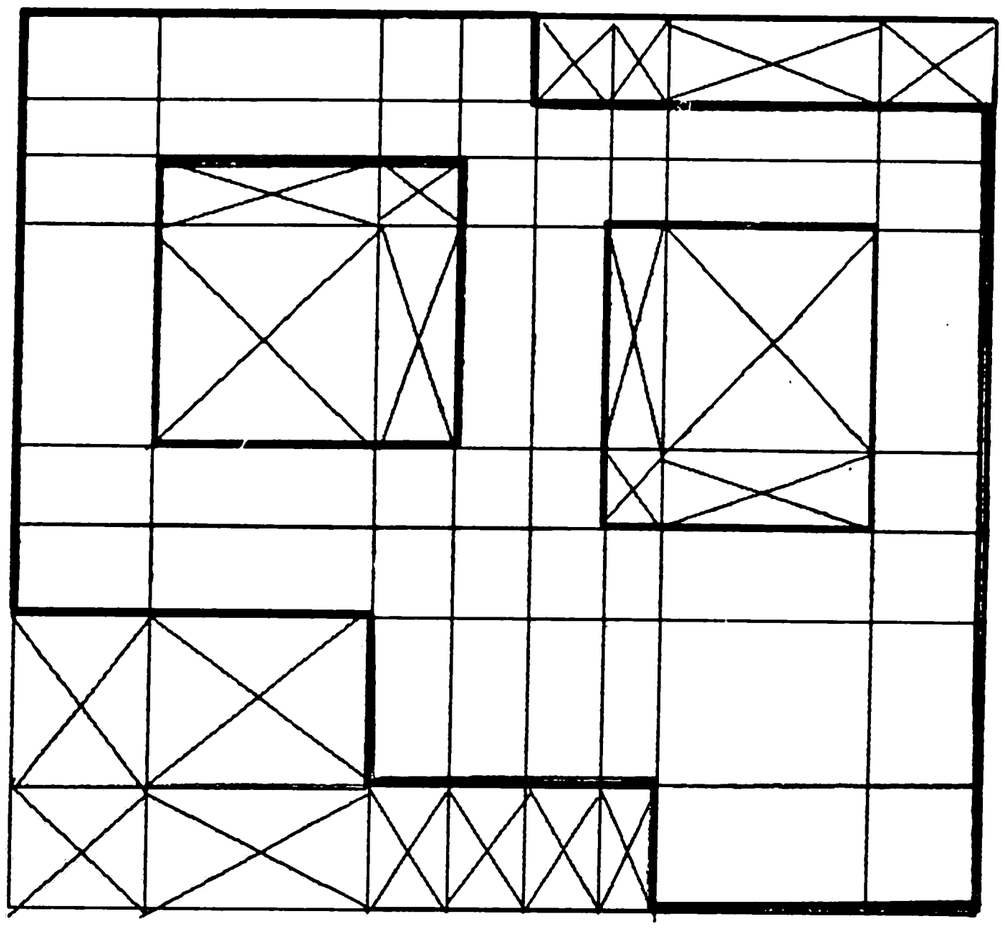


Figure 3-6: Grid created by partitioning layout of Fig. 3-5

If there is an overlap or alignment of the obstacles, the number of distinct edges is reduced. Since the number of nodes generated in the graph is closely related to the number of distinct edges, alignment or overlapping reduces the computational complexity of the algorithm presented here. Chap. 4 discusses the effect of generally orderly layouts on the computation time required in greater detail.

Chapter 4
Simulations and Other Results

This chapter presents a synopsis of the simulations done for both the navigation and path planning algorithms. The first section simulates the ideal path a vehicle would take given a (primary) path while the second section shows the automatic selection of a path given only beginning and ending points.

4.1 Simulating the Drive routine

Fig. 4-1 shows a set of points (marked by +) that may be initially supplied to the vehicle to traverse through. It also shows the secondary points (marked with a •). that are computed through the method of Sec. 2.2.2 and superimposed on the primary path. Fig. 4-2 shows the path taken by the vehicle as it follows secondary points using the technique developed in Sec. 2.2.1 The simulation performed here ignores factors such as wheel slippage, sluggish steering response, friction, and centrifugal forces on the vehicle. In a realistic situation, the vehicle will deviate from the ideal path. However, the strategy described in Sec. 2.2.3 will automatically correct for these errors by dynamically adjusting subsequent arcs of the path. Fig. 4-3 shows the results of a simulation where the vehicle does not follow the ideal path because of a time delay in the steering response. Notice that straight line paths are fine while curved paths deviate from the ideal path. Fig. 4-4 shows another set of points that may be initially supplied to the vehicle to traverse through. Fig. 4-5 shows the ideal path taken by the vehicle as it follows the secondary points.

From these results it is clear that the algorithms developed in Chap. 2 are sufficient to produce a smooth path passing through specified points. The

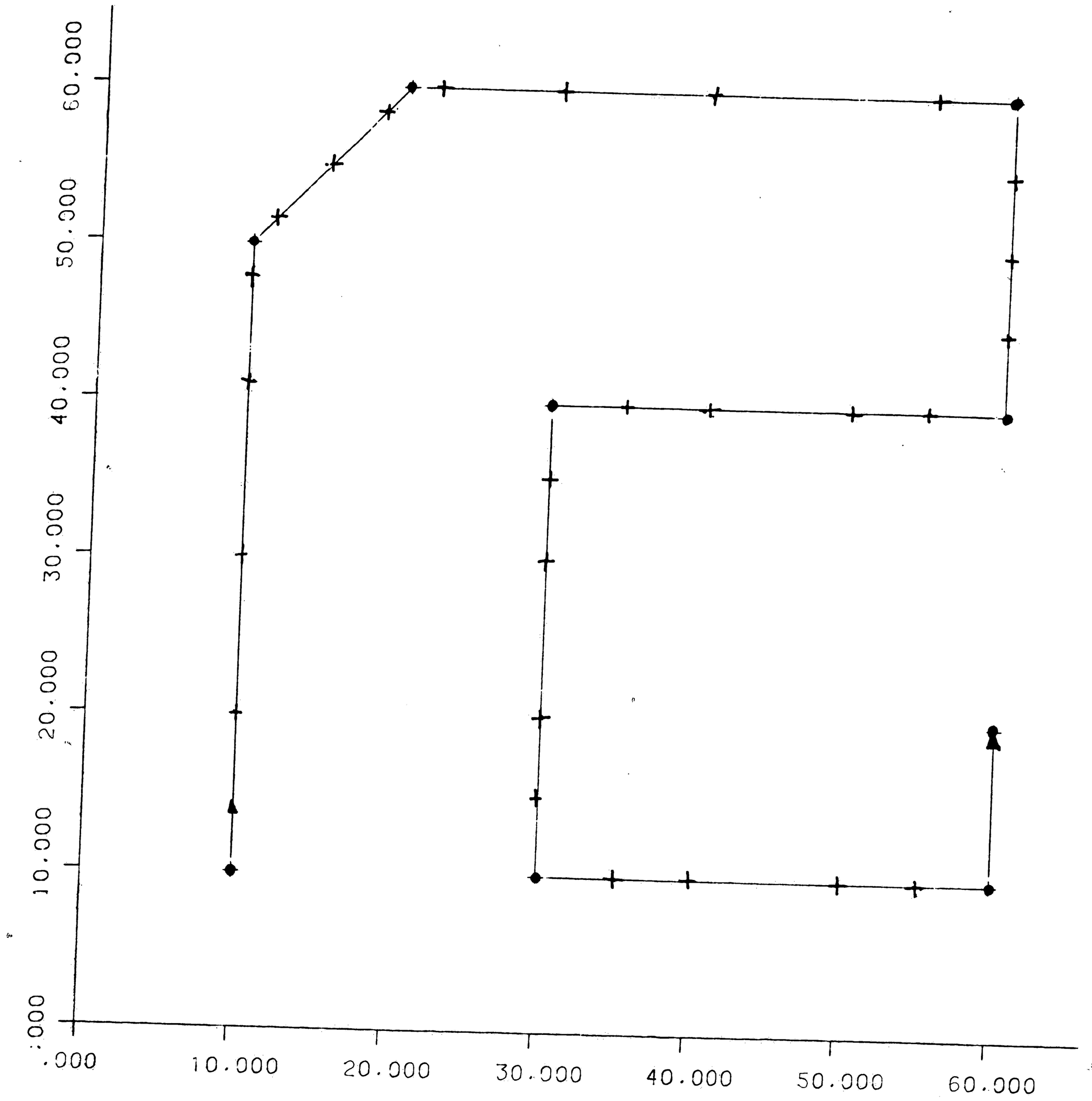


Figure 4-1: Primary and Secondary points on a specified path

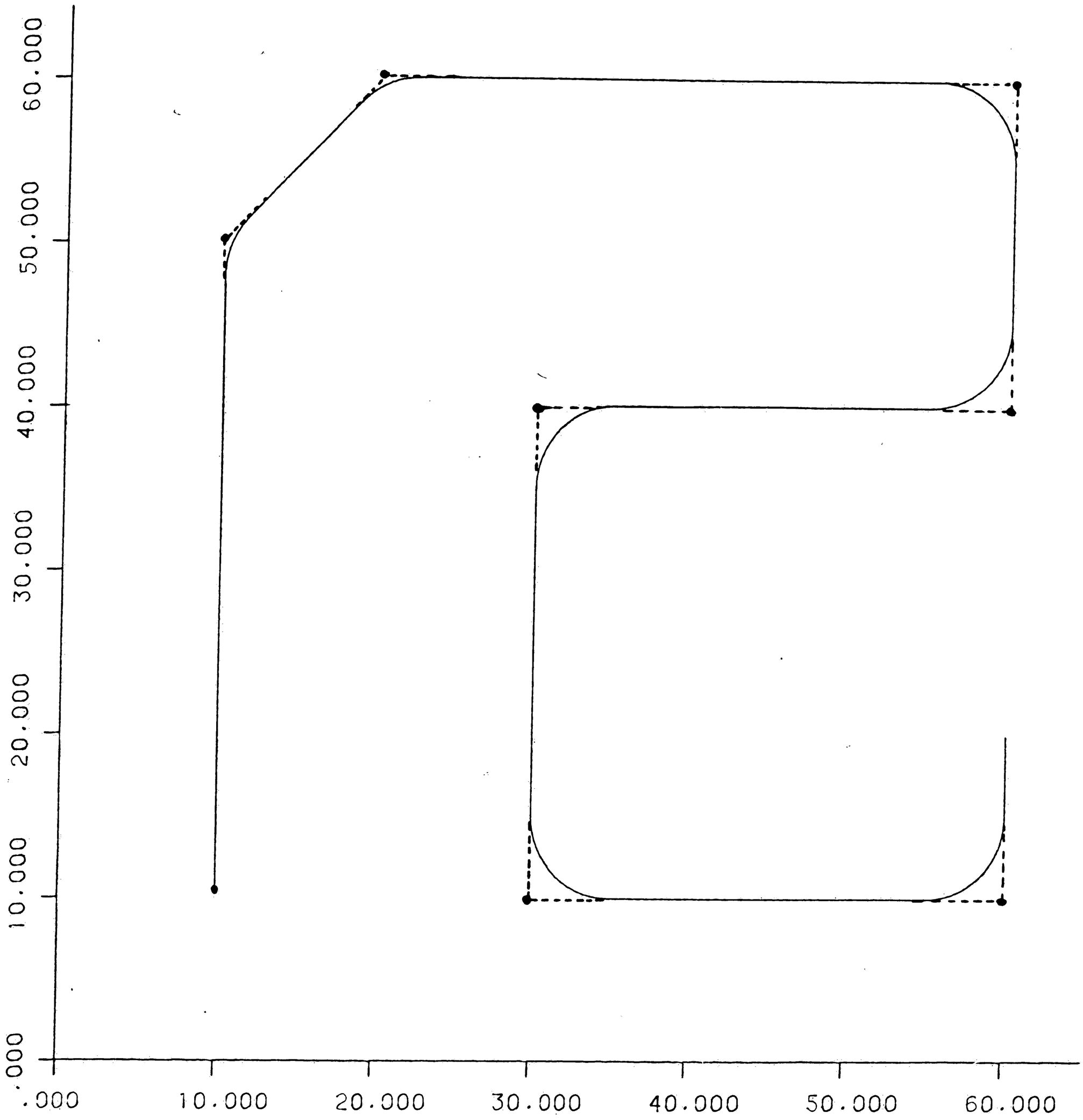


Figure 4-2: Ideal Path taken for the points specified in Fig. 4-1

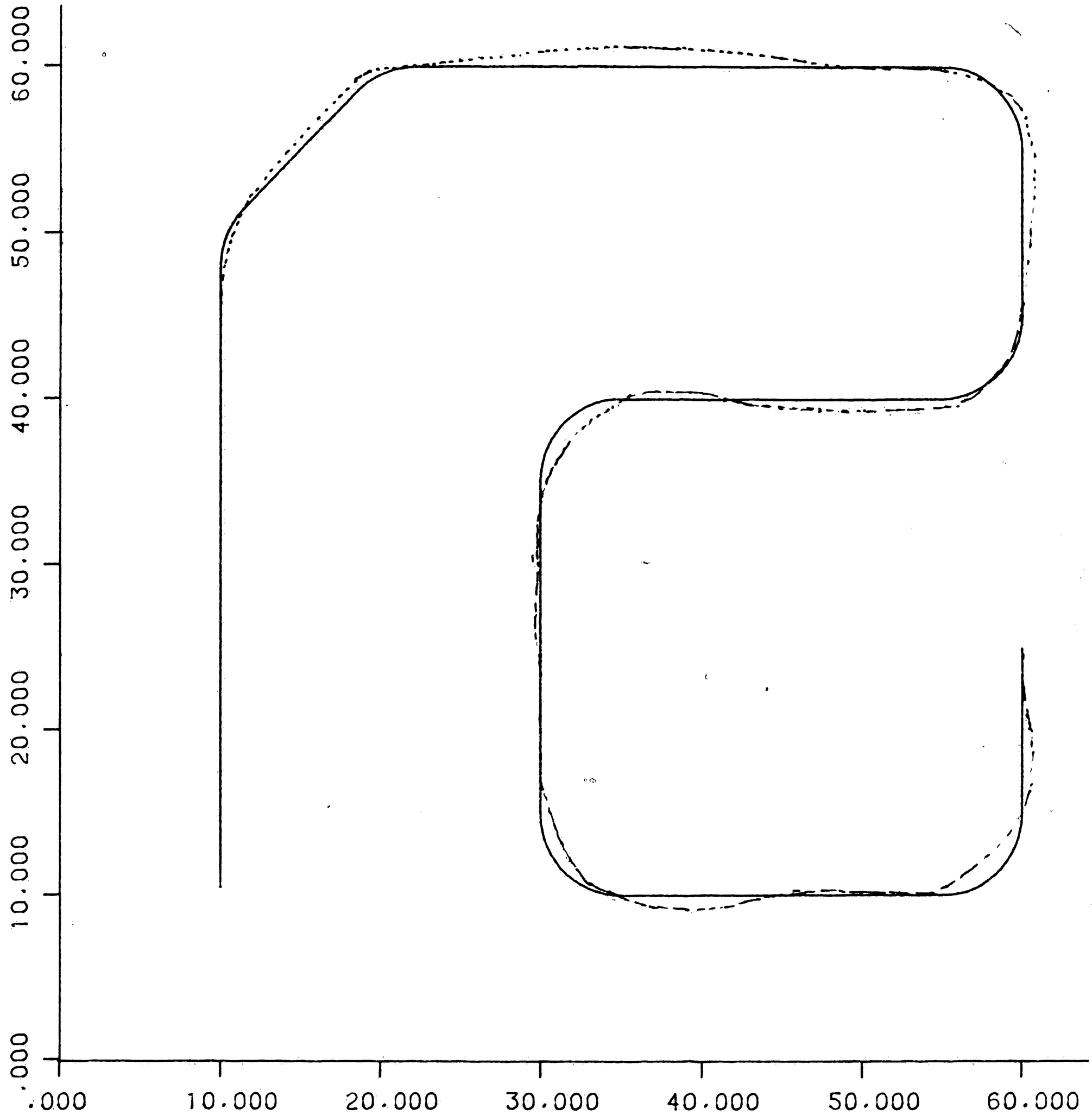


Figure 4-3: Deviations from the ideal path and compensations

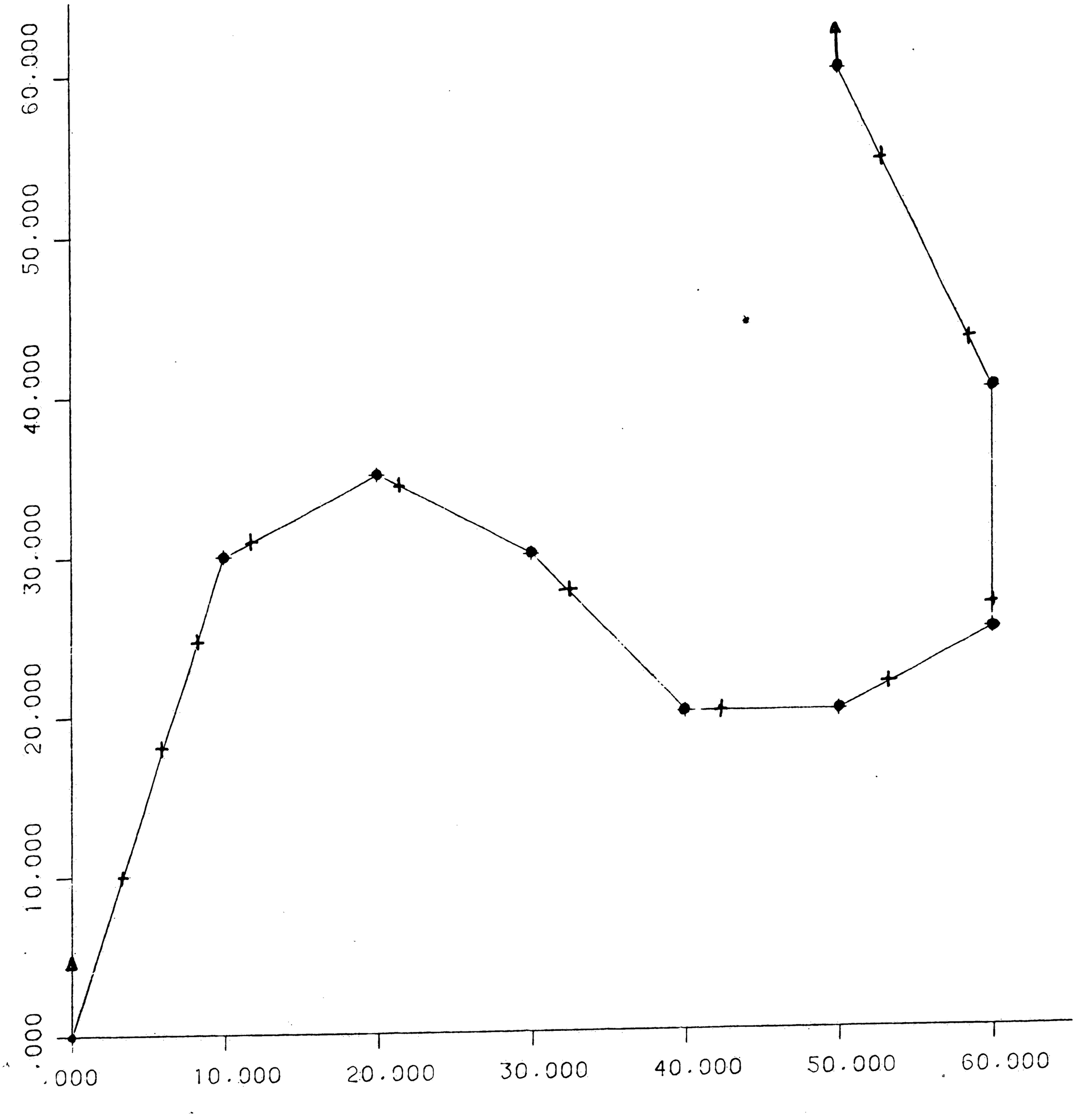


Figure 4-4: Primary and Secondary points on a specified path

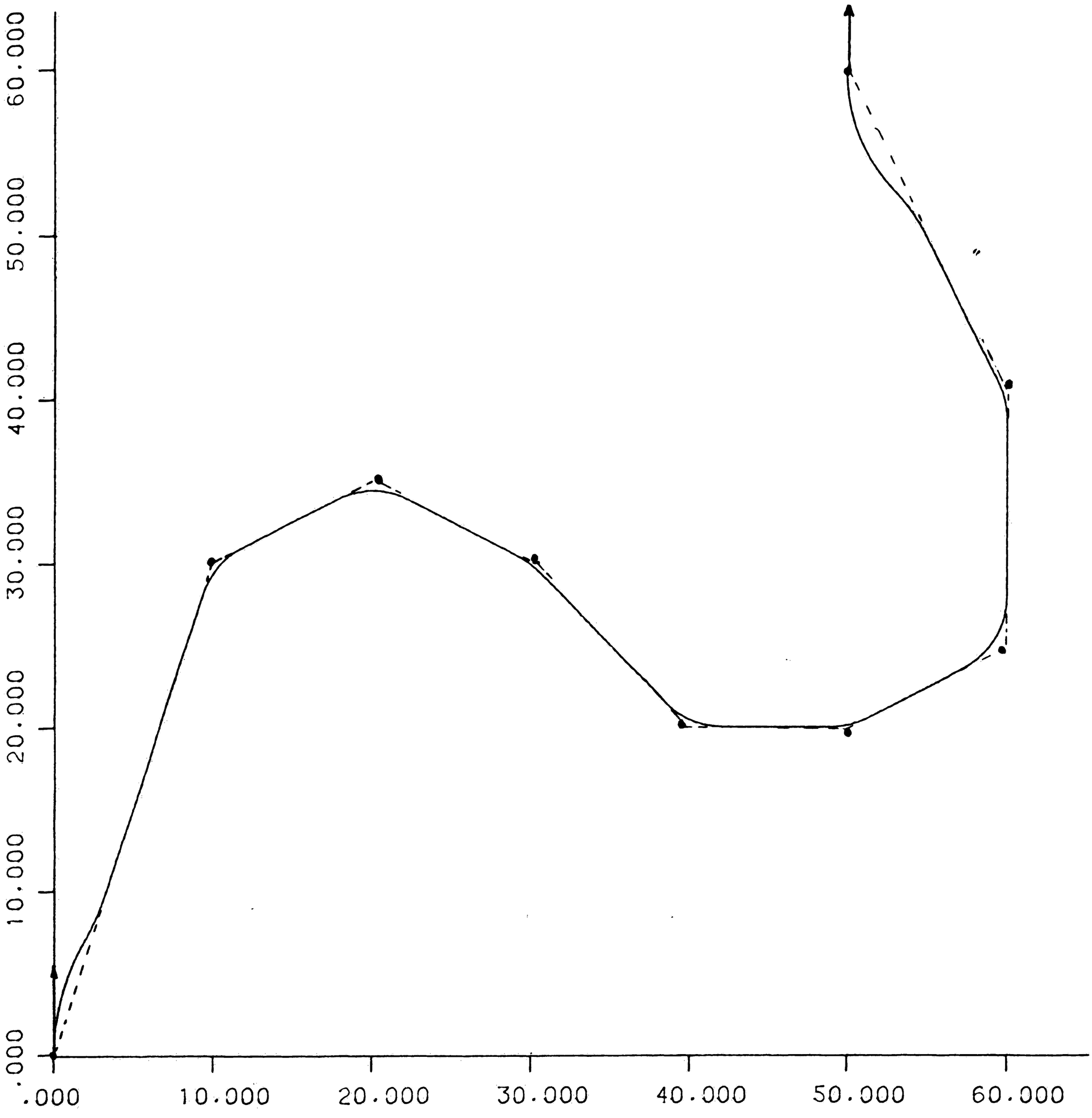


Figure 4-5: Ideal Path taken for the points specified in Fig. 4-4

strategy of compensating for vehicle dynamics at discrete points on the path is successful in keeping the vehicle from straying too far away from the ideal path.

4.2 Simulating the Path Planning

The algorithm described in Chap. 3 was applied to several layouts containing 3 obstacles and one layout containing 4 obstacles. As it has been mentioned before, the computational complexity of finding paths in an environment is proportional to the number of distinct edges in that environment. Correspondingly the best case (that is not trivial) is one where the edges line up exactly (Fig. 4-6, Fig. 4-10). Fig. 4-8 shows the worst case[5] possible for 3 obstacles. It can be seen that when this situation is altered even slightly (Fig. 4-9), the complexity associated with the environment decreases significantly.

For each layout the number of nodes and arcs in the graph generated are presented. 5 sample paths were computed for each layout. Further, a table showing the number of starting and ending nodes, the deviation from the optimal path and the required computation time[6] is also presented.

---

[5] "worst" and "best" refer to the computational complexity for a given environment. Greater complexity results in a larger amount of time to find a successful path.

[6] These times should only be used for between paths because the numbers shown correspond to elapsed "real" time on a mainframe computer [DEC 2065] and not CPU time required.

Table 4-1: Comparison of paths in the "best case": 3 obstacles

Grid Size: **3 x 7**
Number of arcs in the graph generated: **8**
Number of nodes (convex shapes) isolated: **6**
Time taken for setting up the graph: 0.091 secs

path	# of source nodes	#of dest. nodes	% deviation from opt.	computation time (secs)
l-e	2	2	0	0.003
b-h	2	2	0	0.066
a-g	2	2	0	0.070
c-i	1	2	0	0.065
l-f	2	2	0	0.071

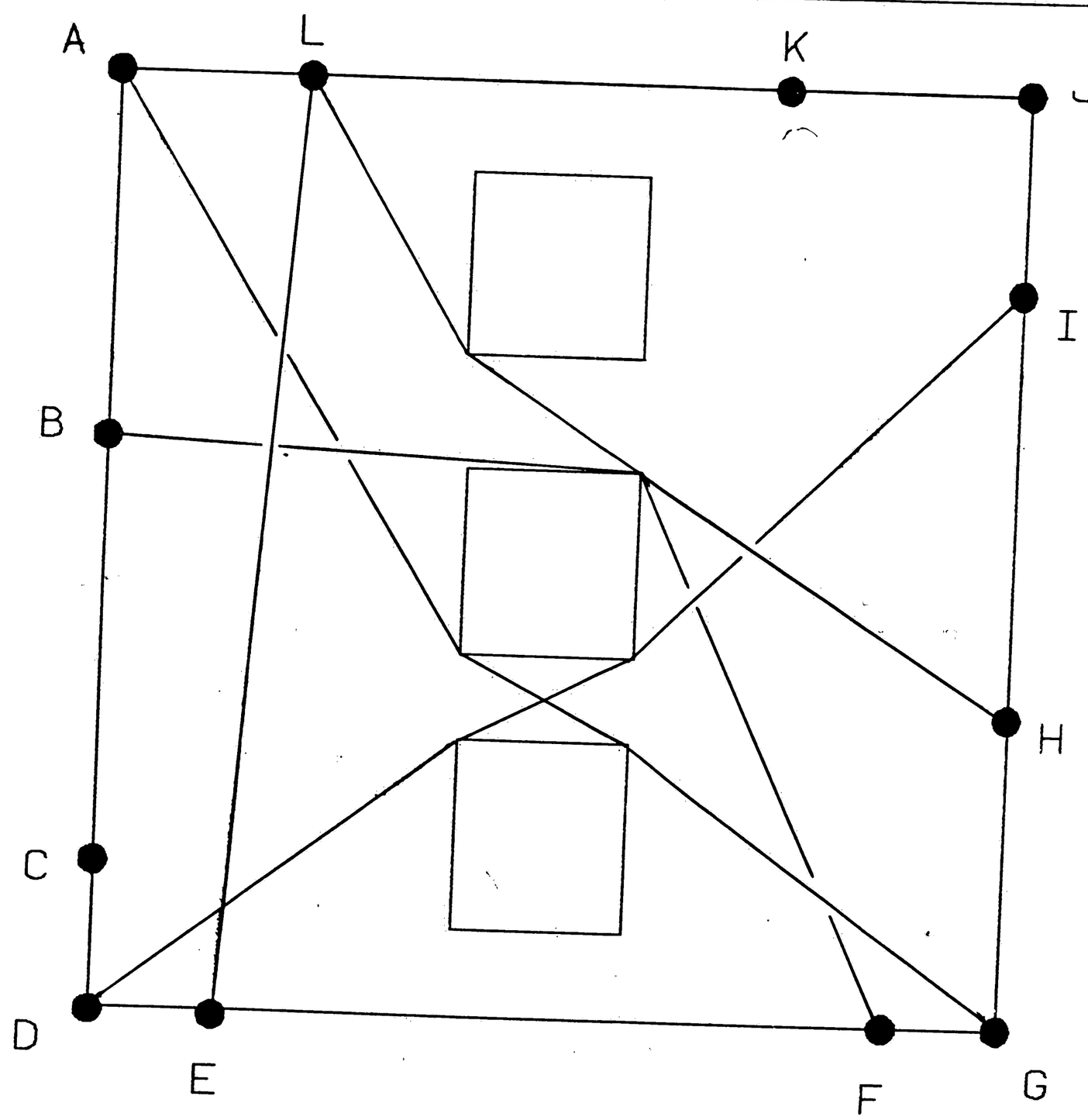


Figure 4-6: Sample paths for "best" case orientation of 3 obstacles

Table 4-2: Comparison of paths in the "average" case: 3 obstacles

Grid Size: **6 x 7**
Number of arcs in the graph generated: **18**
Number of nodes (convex shapes) isolated: 9
Time taken for setting up the graph: 0.499 secs

path	# of source nodes	#of dest. nodes	% deviation from opt.	computation time (secs)
l-e	2	3	0.0	0.003
b-h	1	1	0.7	2.190
a-g	2	3	0.0	0.508
c-i	2	1	0.05	1.578
l-f	2	3	0.8	0.505

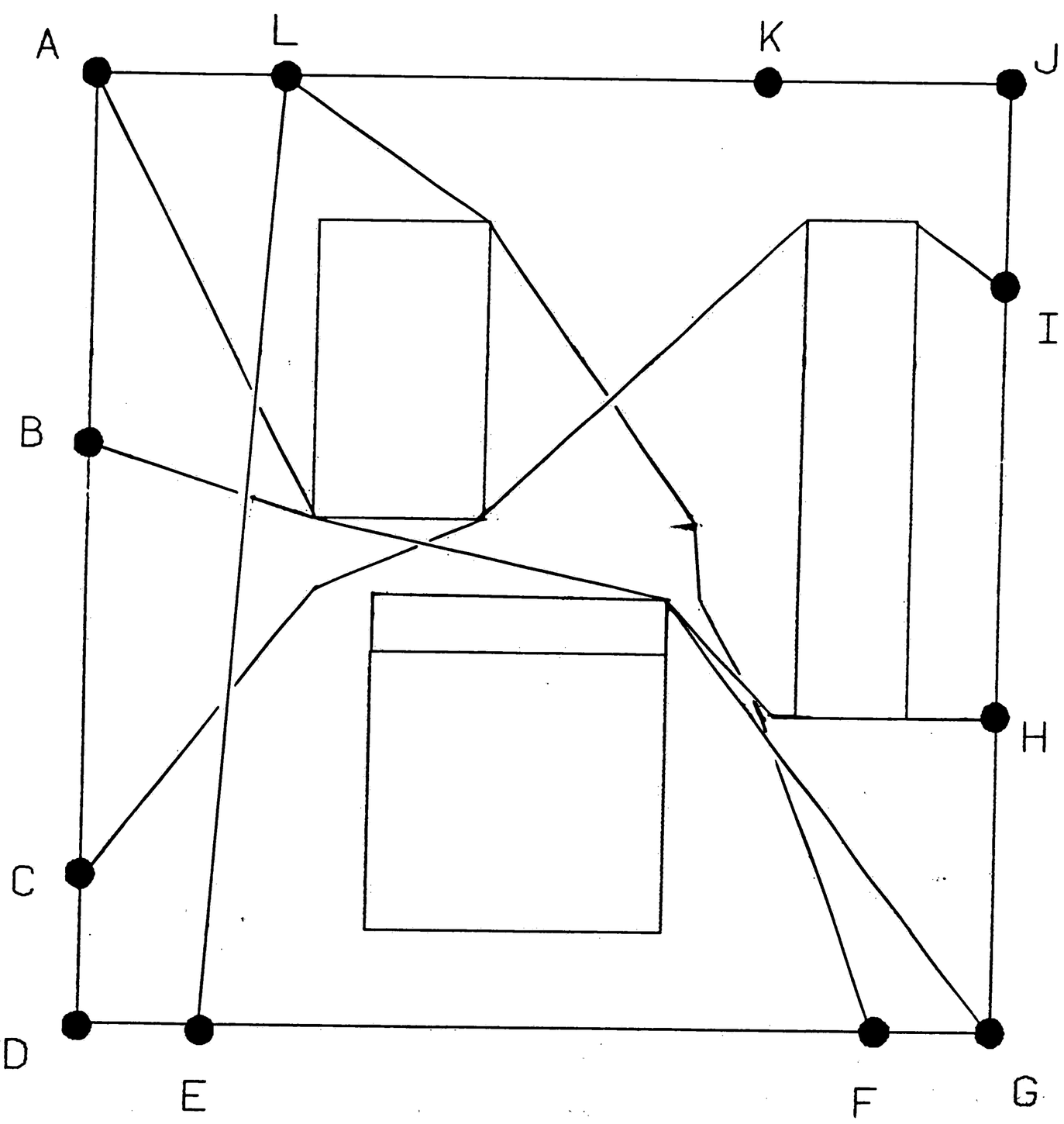


Figure 4-7: Sample paths for "average" case orientation of 3 obstacles

Table 4-3: Comparison of paths in the "worst" case: 3 obstacles

Grid Size: **7 x 7**
Number of arcs in the graph generated: **44**
Number of nodes (convex shapes) isolated: **13**
Time taken for setting up the graph: 0.765 secs

path	# of source nodes	#of dest. nodes	% deviation from opt.	computation time (secs)
l-e	2	3	0.0	0.003
b-h	4	3	0.0	23.684
a-g	2	3	0.03	851.266
c-i	2	2	0.0	320.903
l-f	2	3	0.0	738.136

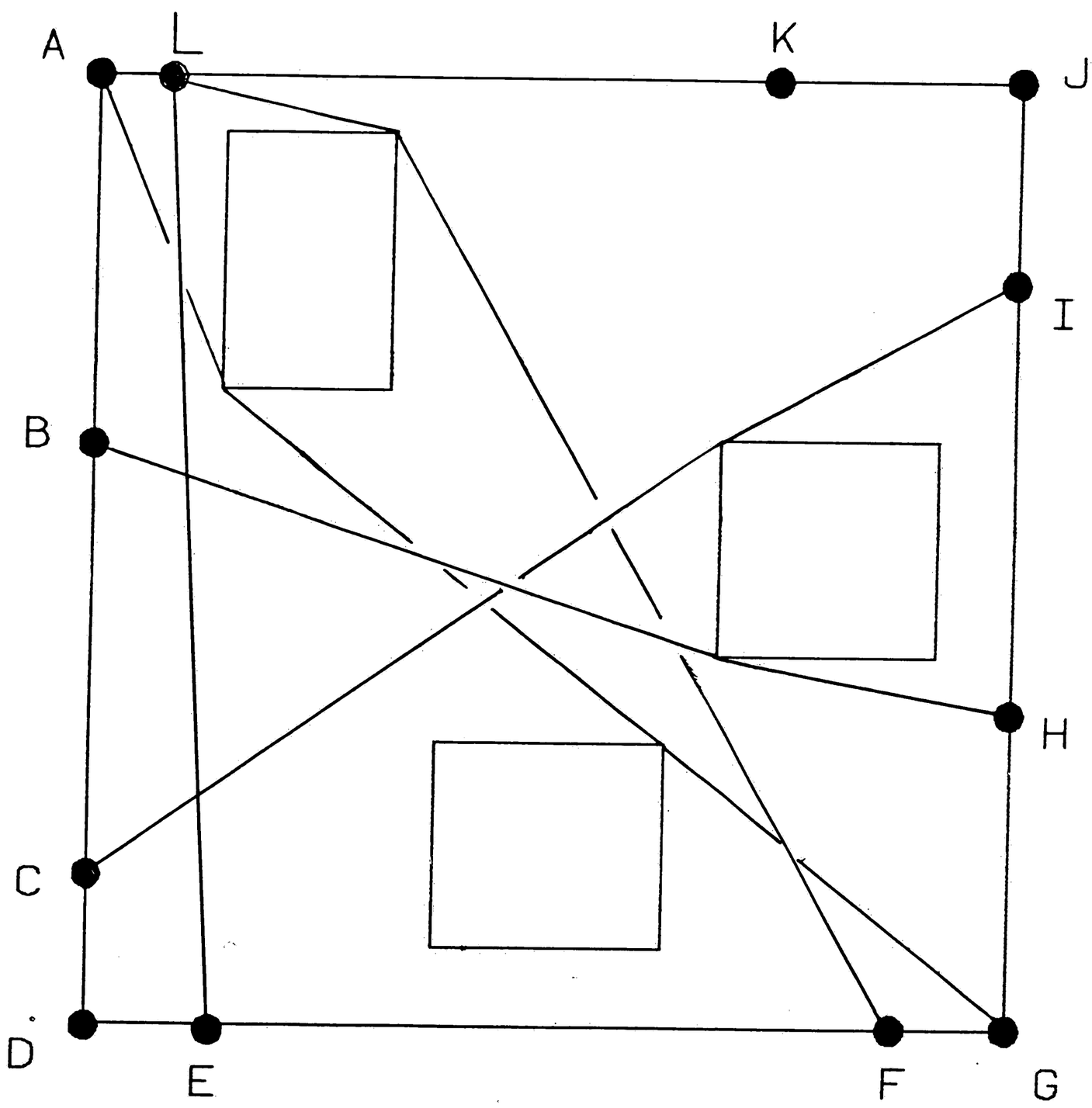


Figure 4-8: Sample paths for "worst" case orientation of 3 obstacles

Table 4-4: Comparison of paths in a "near worst" case: 3 obstacles

Grid Size: **7 x 7**
Number of arcs in the graph generated: **37**
Number of nodes (convex shapes) isolated: **12**
Time taken for setting up the graph: 0.645 secs

path	# of source nodes	#of dest. nodes	% deviation from opt.	computation time (secs)
l-e	2	3	0	0.003
b-h	4	3	0	5.579
a-g	2	3	0	129.470
c-i	2	2	0	77.619
l-f	2	3	0	128.113

The results of this simulation shows that in all scenarios, the time required to set up the graph is very small. Since the algorithm used (described in Sec. 3.2) first isolates horizontal strips, uses these to generate s groups and goes through s-1 iterations, its complexity is $\mathbf{O}(ks^2)$. The term k represents the contribution to the complexity of the number of elements in the group and therefore depends upon the number of vertical edges. This heavy dependence of the time taken to set up the graph on the number of horizontal strips may be illustrated by simulating an environment identical to one in Fig. 4-10 but rotated through 90 degrees. Now the number of horizontal strips goes down from 9 to 3 an consequently the setup time drops from 1.436 secs to 0.321 secs.

These results also show that in most cases the paths obtained are optimal. Deviations from optimality (less than 0.8%), observed in 4 out of 25 sample paths are a result of the inadequacy of only one node look ahead.

It is noticed that the algorithm as presented in Chap. 3 is not adequate enough to handle situations where there is a large number of nodes and arcs in

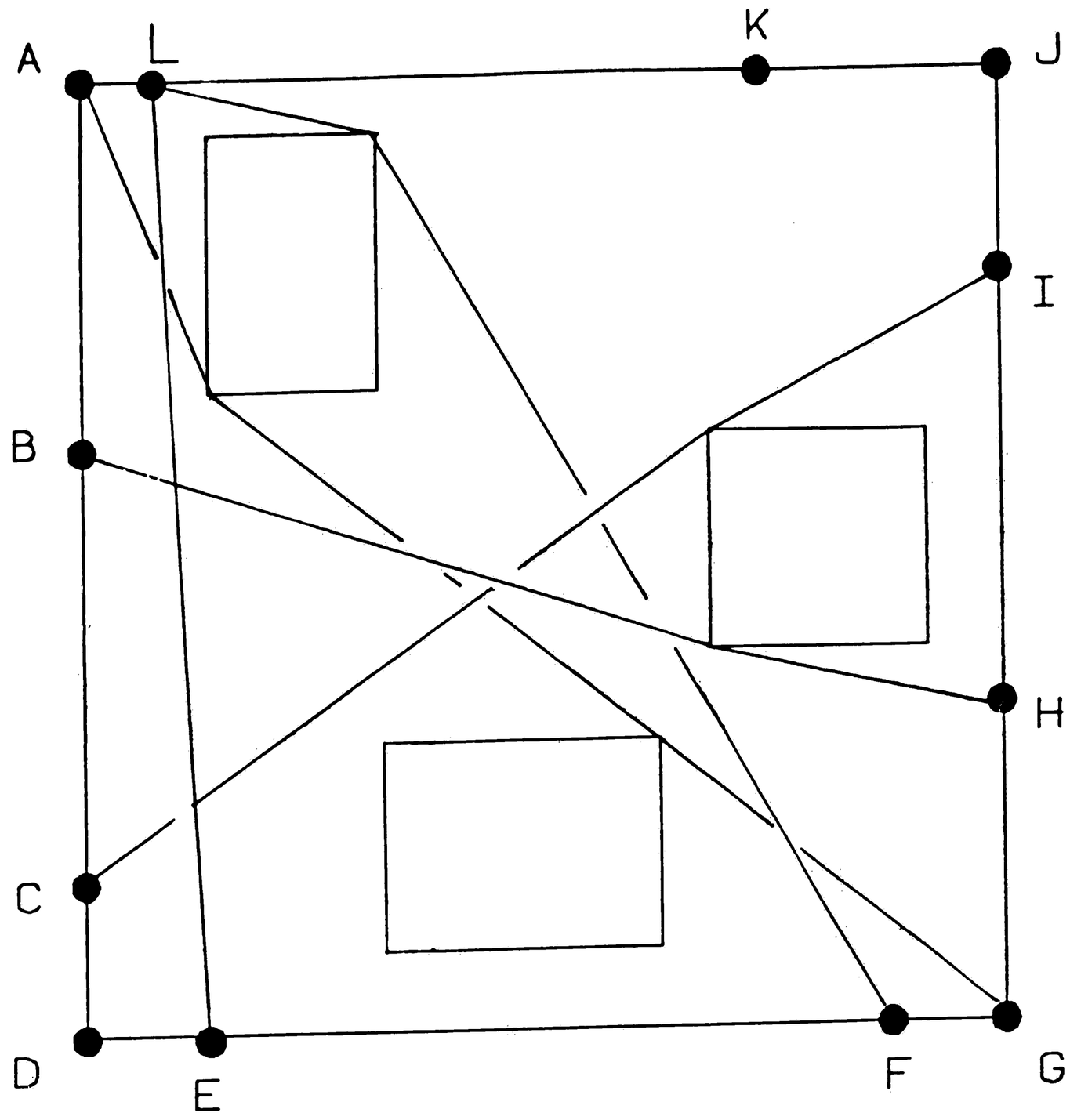


Figure 4-9: Sample paths for "near worst" orientation of 3 obstacles

the graph, in a reasonable amount of time. This is because the algorithm as it stands, necessitates repeated searches of large parts of the graph. The reasons for this and a proposed cure are suggested in the next chapter.

Table 4-5: Comparison of paths in the "best" case: 4 obstacles

Grid Size: **3 x 9**
Number of arcs in the graph generated: **10**
Number of nodes (convex shapes) isolated: **7**
Time taken for setting up the graph: 1.436

path	# of source nodes	#of dest. nodes	% deviation from opt.	computation time (secs)
l-e	2	2	0	0.003
b-h	1	1	0	0.253
a-g	2	2	0	0.199
c-i	1	2	0	0.193
l-f	2	2	0	0.189

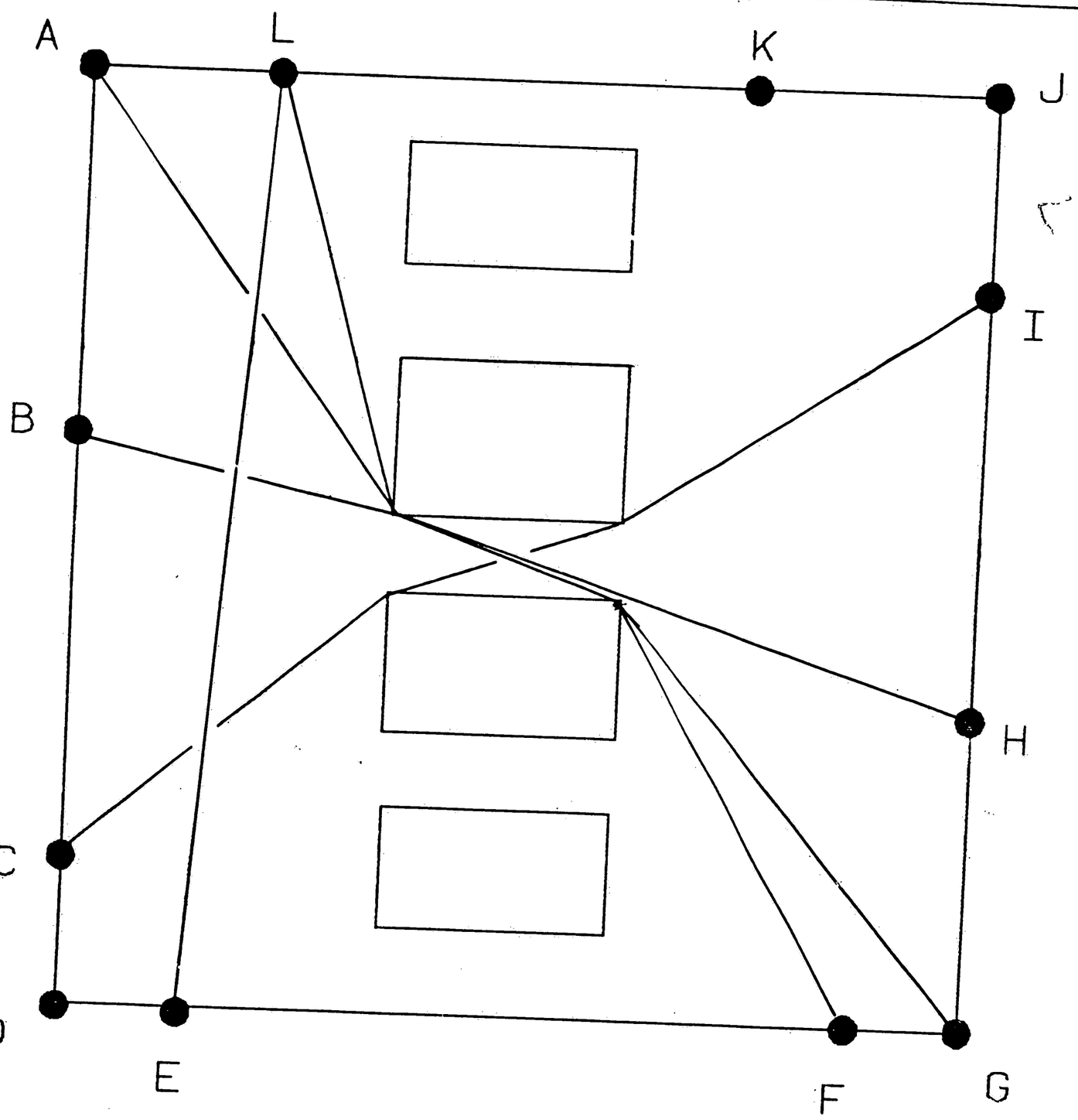


Figure 4-10: Sample paths for "best" case orientation of 4 obstacles

Chapter 5
Conclusions

The navigation scheme described in Chap. 2 has been simulated and tested in several parts and has been found to work well. The most important note to make about the self location methods used is that there are two interdependent systems that together provide an accurate position for the vehicle. The Ground Navigator keeps track of approximate vehicle position and using this information, the Goniometer provides an accurate "fix". The Goniometer reading in turn allows one to establish and remove the the error bias built up in the Ground Navigator. In theory it is possible to get a goniometer reading twice a second but in reality it will only be necessary to get a "fix" at a frequency that will limit the Ground Navigator error bias to threshold[7] or else the Goniometer will not be able to provide an accurate "fix" and the navigation system will fail.

Two parts of the navigation system have been simplified for the prototype vehicle. The first is the correction for movement. At present this is a first order approximation of the vehicle's path while the beacons are being read (it assumes the vehicle is moving in a straight line). In the future a better model of the vehicle's response could be used. Additionally, speeding up the rotation of the goniometer would correspondingly reduce the error in angles at which beacons are detected. For example, if the vehicle is traveling at velocity v, the time taken for one Goniometer rotation is t, and if the vehicle is moving such that it equidistant from a beacon (a distance d away) at the beginning and end

---

[7] to be exactly determined experimentally; rough estimates put the maximum allowable bearing error at about 5 degrees.

of the rotation, then the maximum error (with out correction) due to movement is: $\frac{t \cdot v}{d}$. If the goniometer rotation is speeded up by a factor of m, the error term is reduced by the same factor. Essentially, this approach would decrease the inaccuracy induced by making a first order approximation of vehicle movement during the time that it takes to obtain a "fix".

The second area that should be improved upon in the future is the dynamic feedback used to control the steering angle while the vehicle is moving along a specified path. At present, position and bearing feedback is obtained at most twice between two specified points on the path. The secondary path (described in Sec. 2.2.2) could be altered so as to generate points closer together effectively reducing the sampling interval at which a compensation is made for the vehicle straying off the path.

The path planning procedure outlined in Chap. 3 has many advantages. It takes into account *all* the largest (rectangular) convex areas in a layout and allows for most straight line path segments to be found very efficiently. In addition, if the source and the destination points belong to the same convex area, the optimal path is picked trivially. Because an overlap of convex areas is permitted, the graph generated is more complex than the one generated by previous researchers. However, in all except the worst case, the number of nodes in the graph is $\mathbf{O}(n)$, where n is the number of obstacles. Preliminary investigation shows that the number of nodes is approximately $6n$-5 in the worst case. Simulations show that the computational complexity of the path planning algorithm is directly proportional to the number of arcs in the graph produced. The less alignment there is among the obstacles, the greater number of convex shapes (and arcs) isolated. Thus the worst case is where none of the

edges overlap or align. A comparison of tables 4-3 and 4-4 shows that even if one of the edges aligns, the number of nodes, the number of arcs, and the amount of time taken to find a path reduces considerably.

The relatively large times taken by the algorithms when searching a large graph with many nodes and arcs can be explained as follows: The algorithm makes no provision for recognizing previously traversed large graph segments if they are approached from a different node than before. For example a node path has developed in the following manner: A → B → C → D → ... Suppose that the path backtracks to node B and then moves to node E. If there is an arc between E and C, then once again the entire graph connected to C is searched. This was felt necessary because approaching a node that has been previously approached, but from another route, can produce a different point path of a different length. As the number of nodes (and arcs) increases, the amount of time taken to search this graph will also rise. To take care of this deficiency, the algorithm presented in Chap. 3 can be modified in the following manner: For every node, keep the cost of the best path from that node to the destination node. This list is added to as the path backtracks from a destination node or a node that belongs to the list. Before a new node is appended to the current path, search this list to check if a successful path from that node to the destination node has been found in the past. If so, then add the cost of that path to the cost of the path determined so far. If this cost exceeds the best cost (of a successful path) by a predetermined margin, then instead of going further, backtrack.

One of the advantages of the method is that a graph can be precomputed (without knowledge of source and destination points) rapidly from a map of the

environment the layout. In order to come up with a near optimal path, we concurrently develop a graph node path as well as an actual point path for robot travel. This permits dynamic cost allocation to the point path and at the same time, exploitation of the backtracking technique for the node path.

This procedure looks ahead only one node in the node path during the graph search. Consequently the path segment $C_{i+1} \rightarrow C_{i+2}$ is determined independently of the segment $C_i \rightarrow C_{i+1}$. It is therefore possible in certain cases that the path obtained $C_i \rightarrow C_{i+1} \rightarrow C_{i+2}$ is not the optimal path from $C_i \rightarrow C_{i+2}$ (when C_i and C_{i+2} can be joined by a straight line). However, this drawback can be overcome by refitting the point path every time a new point is added to the path. Path points will not fall on corners of inadmissible areas only when the one node look ahead is not enough resulting in a nonoptimal path.

The work presented in Chap. 3 is limited in considering only rectangular convex areas. There are an infinite number of non-rectangular areas that exist in any given environment. However, if some intelligently chosen non-rectangular prime convex areas are added to the set of convex areas, the optimality of the path obtained will improve. Such a choice would have to be based on a statistical study of paths frequently traveled. More paths would be found trivially as well as there would be a less chance of finding a "bent" path where a straight line path were possible. In addition, a simple preprocessing procedure could be included to quickly determine if a direct straight line path exists between the given pair of source and destination points. This would be particularly useful if the path is oblique and would normally require graph traversal as well as the refitting explained above. Because obstacles have been

extended beyond their physical dimensions (to avoid collisions with finite sized robots), a postprocessing procedure would be required to advance the robot from the artificial boundary of the obstacle to the physical docking bay (if there were such a need).

In a sense, the modification to the graph searching technique mentioned above incorporates the notion of "learnt" behavior- repeated graph searches will occur less and less as successful paths are found. One more modification can be made in the same spirit: Procedure **findnewnode** in Sec. 3.4 can be configured to look for the best possible next node to approach the present destination node, rather than the first valid one. This can be done by keeping track of how many times each link of each node was chosen in past successful paths on the way to a given destination. This approach would serve to cut down the graph search considerably because there would be a greater chance that the best paths would be found early in the search. Thus less time would be spent looking at entire paths with costs greater than the best cost. It should be noted, however, that even though this learning capability would improve the computational speed, it would be expensive in terms of its memory requirements.

References

1. S. Singh and M. D. Wagh, "Robot Path Planning Using Intersecting Convex Shapes", Submitted to IEEE Journal of Robotics and Automation for Review

2. L. Gouzenes, "Strategies for solving Collision Free Trajectory Problems for Mobile and Manipulator Robots". *The International Journal of Robotics Research*, Vol. 3, pp.51-65, Winter, 1984.

3. R. Chatila and J. P. Laumond, "Position Referencing and Consisten World Modeling for Mobile Robots", *IEEE Conference on Robotics and Automation*, St. Louis, MO, March 1985.

4. T. Muller, *Automated Guided Vehicles*, Springer Verlag, New York, NY, 1983.

5. S. Singh, CSEE Dept., Lehigh University, *Documentation of Software, Hardware and Design Ideolgies Used in the Development of CYCLOPIAN: A Prototype Automated Guided Vehicle System*, 1985, A supplemental Report submitted to SI Handling

6. M. B. Ignat'yev, F. M. Kulakov and A. M. Pokrovsky, "Robot Manipulator Control Algorithms". Tech. report JPRS 59717, NTIS, August 1973.

7. R. A. Brooks, "Solving the Find-Path Problem by Good Representation of Free Space", *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-13, pp. 190-197, March, 1983.

8. R. A. Brooks and Lozano-Perez, Tomas, "A Subdivision Algorithm in Configuration Space for Findpath with Rotation", *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-15, pp. 224-233, March, 1985.

9. T. Lozano-Perez and M. A. Wesley, "An Algorithm for Planning Collision-Free Paths Among Obstacles", *Communications of ACM*, Vol. 22, pp. 560-570, October, 1979.

10. J. L. Crowley, "Navigation for an Intelligent Mobile Robot", Tech. report CMU-RI-TR-84-18, Carnegie-Mellon Univ., August 1984.

11. R. Chatila, "Path Planning and Enviornment Learning in a Mobile Robot System", *European Conference on Artificial Intelligence*, Orsay, France, July 1982.

12. D. T. Kuan, J. C. Zamiska and R. A. Brooks, "Natural Decomposition of Free space for Path Planning", *IEEE Conference on Robotics and Automation*, St. Louis, MO, March 1985.

13. M. M. Mano, *Digital Design*, Prentice Hall Inc, Englewood Cliffs, NJ, 1984.

14. E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*, Computer Science Press, Rockville. MD. 1984.

15. D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, Addision-Wesley Publishing. Reading,MA, Vol. 1, 1968.

Vita

Sanjiv Singh was born on March 3, 1962 to Mr. & Mrs. Surindar Singh in Barelli, India. Having done most of his schooling at the Cathedral and John Connon High School in Bombay, India, he moved to Ft. Collins, Colorado for the last year of high school. He recieved his Bachelors degree at the University of Denver in 1983 in Computer Science and completed a year in a Masters program at the same institution before transferring to Lehigh University in the Fall of 1984. He has worked first as a Teaching Assistant, and then as a Research Assistant in the Computer Science and Electrical Engineering Department.

After obtaining his Masters degree at Lehigh, he has joined the Robotics Institute at Carnegie-Mellon University as a Research Engineer.